«a2» 2012.01.01

Volume V

os16

TXT
htm
PDF
pdf
P1
P2
P3
P4
epub

```
os16: a basic os
This is terminal /dev/console0
login:  root
password:  ****
```

ISBN 978-88-905012-2-7

Via Morganella Est, 21 -- I-31050 Ponzano Veneto (TV) -- ⟨*appunti2 (ad) gmail·com* ⟩

L'opera, nei file sorgenti e nella composizione finale, include degli esempi in forma di sequenze animate, contenenti eventualmente anche delle spiegazioni vocali. Si tratta di video brevi e di qualità molto

bassa. Tuttavia, a seconda di come viene diffusa o fruita l'opera, può darsi che sia necessario assolvere a degli obblighi di legge.

Il numero ISBN 978-88-905012-2-7 si riferisce all'opera originale in formato elettronico, pubblicata a titolo gratuito.

Per tutti i riferimenti dell'opera si veda ⟨*http://informaticalibera.net*⟩; per essere ridiretti automaticamente all'origine della distribuzione in-linea, si può usare l'indirizzo ⟨*http://appunti2.net*⟩. Al momento della pubblicazione di questa edizione, i punti di distribuzione in-linea più importanti, sono presso Internet Archive (⟨*http://www.archive.org/details/AppuntiDiInformaticaLibera*⟩), il GARR (⟨*http://appuntilinux.mirror.garr.it/mirrors/appuntilinux/*⟩) e ILS (⟨*http://appunti.linux.it*⟩). Tuttavia, per motivi tecnici, l'edizione pubblicata in-linea potrebbe non essere completa di tutti i contenuti, oppure potrebbe non essere sempre aggiornata. Pertanto, chi desidera ricevere **gratuitamente** una copia dell'ultima edizione di «a2» in DVD-ROM può scrivere all'autore, specificando l'indirizzo postale a cui va spedito il materiale:

daniele giacomini
via Morganella EST, 21
31050 Ponzano Veneto (TV)
⟨*appunti2 (ad) gmail·com*⟩

Dal momento che il pacchetto dei sorgenti dell'opera è molto grande (SGML, compilabile con Alml), se si desidera ricevere anche questo, è bene specificarlo.

Successivamente all'edizione 2009.11111 dell'opera (21 luglio 2009), inizia un processo di razionalizzazione con l'eliminazione di molti contenuti non più attuali o non più utili. Chi desidera ricevere una copia della versione 2009.11111 o di versioni precedenti, può farne ugualmente richiesta.

Progetto

Programmare a 16 bit

TXT htm PDF pdf P1 P2 P3 P4 epub

89.1 Microprocessori x86-16

I microprocessori x86-16 sono sostanzialmente costituiti dal 8086 e dal 8088, con la caratteristica di gestire registri a 16 bit e di poter indirizzare complessivamente fino a 1024 Kibyte, suddividendo però la memoria in segmenti da 64 Kibyte. Questa famiglia ha il limite di disporre di pochi registri per usi generali, spesso vincolati a un ruolo preciso, nell'ambito di certe istruzioni.

Dal momento che esiste una grande quantità di modelli di microprocessori compatibili con la vecchia famiglia a 16 bit e dato che sono disponibili simulatori ed emulatori, può essere ancora interessante lo studio della programmazione a 16 bit, riferita al modello x86-16, se non si devono affrontare problematiche relative alla protezione della memoria e a gestioni sofisticate della stessa.

Questo e gli altri capitoli dedicati alla programmazione con i microprocessori x86-16 e l'architettura dell'elaboratore IBM PC dei primi anni 1980, si limitano ad affrontare le questioni che consentono di lavorare con i registri di segmento posti tutti allo stesso valore (salva la possibilità di travasare dei dati da una parte della memoria all'altra). Molte questioni importanti non vengono affrontate e si rimanda ai riferimenti posti alla fine dei capitoli, per gli approfondimenti eventuali, oltre che al capitolo 70, in cui si fa riferimento ai microprocessori x86-32.

89.1.1 Segmenti

Prima di considerare i registri di un microprocessore x86-16, è importante comprendere il concetto di ***segmento***, utilizzato in questo contesto.

Dal momento che i registri sono a 16 bit, con questi si possono rappresentare valori senza segno da zero a 65535; pertanto, dato che la memoria è organizzata in byte, con un registro si può scandire soltanto un intervallo di 64 Kibyte. Per poter scandire lo spazio di 1024 Kibyte, occorrono due registri, in modo da comporre assieme un indirizzo da 20 bit.

Per indirizzare la memoria, a qualunque titolo, nei microprocessori x86-16 è necessario un ***registro di segmento*** e un altro valore che esprima lo scostamento dall'inizio del segmento a cui il contesto si riferisce.

I segmenti possono collocarsi in memoria con una certa libertà, pertanto possono sovrapporsi, parzialmente o completamente. In pratica la memoria viene suddivisa idealmente in ***paragrafi*** (o *click*) da 16 byte ciascuno e i segmenti possono iniziare

soltanto all'inizio di un paragrafo. Per garantire che ciò avvenga in questo modo, i registri che sono dedicati a rappresentare l'inizio di un segmento, riportano il numero del paragrafo, ovvero l'indirizzo assoluto di memoria diviso per 16. Questa divisione si ottiene con un semplice scorrimento a destra di quattro bit; pertanto, per ritrovare il valore originale è sufficiente fare lo scorrimento opposto, verso sinistra. Per esempio, il valore $159D_{16}$ contenuto in un registro di segmento, individua in realtà l'indirizzo $159D0_{16}$, pari a 88528_{10}.

Come accennato, per individuare una certa posizione in memoria si usa sempre un registro di segmento e un altro valore che rappresenta lo scostamento a partire dall'inizio del segmento a cui si riferisce il contesto. Per esempio, se il registro di segmento contiene il valore $159D_{16}$ e si specifica lo scostamento $FFFE_{16}$, si sta in pratica facendo riferimento alla posizione di memoria $159D0_{16}+FFFE_{16}$, pari a $259CE_{16}$, ovvero 154062_{10}.

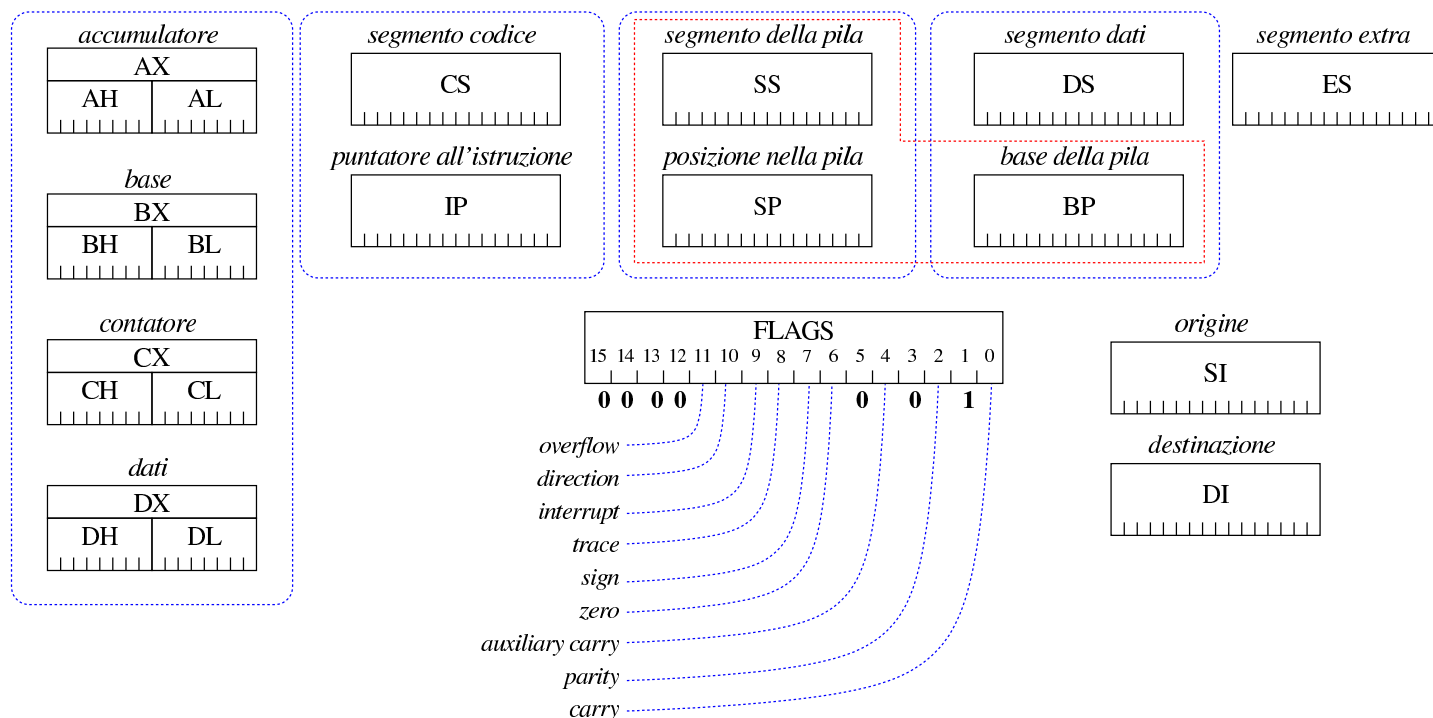
Stante questa organizzazione, per indicare in un documento un certo indirizzo di memoria, si può usare la definizione di «indirizzo efficace» e si può scrivere un solo numero, come per esempio $159DF_{16}$.

Riquadro 89.1. Valori affiancati e divisi dal simbolo ':'.

Nella programmazione a 16 bit, con i microprocessori della famiglia x86, per affiancare due valori a 16 bit si usa normalmente il segno di due punti, come per esempio ***DX***:***AX*** o $159D_{16}$:$FFFE_{16}$.
In generale, questa rappresentazione indica soltanto che si vuole fare riferimento a un numero a 32 bit, formato dall'unione delle due parti indicate, ma il significato che questo numero deve avere va interpretato in base al contesto. Per esempio, ***DX***:***AX*** potrebbe essere il risultato di una moltiplicazione, da prendere numericamente tale e quale, nel senso che il registro ***DX*** rappresenta i 16 bit più significativi; ma in un altro contesto, ***DS***:***SI*** può fare riferimento a un indirizzo che si interpreta come ***DS***·16+***SI***. Nello stesso modo, il numero rappresentato come 1000_{16}:$59DF_{16}$, potrebbe indicare precisamente il valore $100059DF_{16}$, oppure, se si tratta di un indirizzo, composto da segmento e scostamento, andrebbe inteso come $159DF_{16}$.

89.1.2 Registri

I registri dei microprocessori x86-16 sono schematizzati dalla figura successiva. I registri per uso generale, denominati ***AX***, ***BX***, ***CX*** e ***DX***, possono essere utilizzati nella loro interezza o divisi in byte; per esempio si può intervenire nel byte meno significativo di ***AX*** con il nome ***AL*** (*low*) e si può accedere al byte più significativo con il nome ***AH*** (*high*).

Figura 89.2. I registri dei microprocessori x86-16.

I registri di segmento sono: ***CS***, ***DS***, ***SS*** e ***ES***. Il segmento individuato dal registro ***CS*** (*code segment*) è quello in cui si svolge il codice in corso di esecuzione, e il puntatore all'istruzione da eseguire, nell'ambito del segmento codice, è contenuta nel registro ***IP*** (*instruction pointer*, ma noto anche come *program counter* e indicato a volte con la sigla «PC»). Il segmento individuato dal registro ***SS*** (***stack segment***) è quello in cui si trova la pila dei dati, ovvero quella struttura che consente il trasferimento delle variabili alle funzioni o la creazione di variabili locali. L'indice della pila è costituito dal registro ***SP*** (*stack pointer*) e l'indirizzo della base della pila, nell'ambito della funzione in corso di esecuzione, viene annotato convenzionalmente nel registro ***BP*** (*base pointer*). Il segmento individuato dal registro ***DS*** (***data segment***) è quello in cui si trovano i dati correnti, mentre il segmento del registro ***ES*** (***extra segment***) riguarda un'area dati alternativa, utile soprattutto quando si vogliono fare dei trasferimenti di dati tra segmenti differenti.

Convenzionalmente è stato adottato il registro ***BP*** per annotare il riferimento all'inizio della pila di una funzione, per poter accedere agli argomenti attuali o alle variabili locali con un riferimento relativo a tale puntatore. Tuttavia, va osservato che il segmento a cui si riferisce il registro ***BP*** è quello dei dati, ovvero ***DS***, per cui, quando si utilizza ***BP*** per accedere al contenuto della pila, è indispensabile che ***DS*** sia uguale a ***SS***.

Il registro ***FLAGS*** raccoglie gli indicatori disponibili, come descritto nella tabella successiva. In alcuni documenti, tale registro è chiamato *program status word* e abbreviato come «PSW».

Tabella 89.3. Gli indicatori principali contenuti nel registro ***FLAGS***.

Indicatore (*flag*)	Bit	Descrizione
C *carry*	0	È l'indicatore del riporto per le operazioni con valori senza segno. In particolare si attiva dopo una somma che genera un riporto e dopo una sottrazione che richiede il prestito di una cifra (in tal caso si chiama anche *borrow*).
1	1	Riservato.
P *parity*	2	Si attiva quando l'ultima operazione produce un risultato i cui otto bit meno significativi contengono una quantità pari di cifre a uno.
0	3	Riservato.
A *auxiliary carry*	4	È un tipo di riporto ausiliario.
0	5	Riservato.
Z *zero*	6	Viene impostato dopo un'operazione che dà come risultato il valore zero.
S *sign*	7	Riproduce il bit più significativo di un valore, dopo un'operazione. Se il valore è da intendersi con segno, l'indicatore serve a riprodurre il segno stesso.
T *trace*	8	Se è attivo, fa in modo che il microprocessore possa funzionare un passo alla volta.
I *interrupt*	9	Se è attivo, le interruzioni hardware sono abilitate, diversamente risultano bloccate.
D *direction*	10	Si usa per automatizzare le operazioni relative alle stringhe. Se è a zero, indica che la scansione della memoria deve procedere incrementando gli indici; se invece è pari a uno, la scansione deve proseguire decrementando gli indici.
O *overflow*	11	È l'indicatore di traboccamento per le operazioni che riguardano valori con segno.
0	12	Riservato.
0	13	Riservato.
0	14	Riservato.
0	15	Riservato.

I registri che sono definiti «per usi generali», hanno comunque un ruolo predominante. Tra questi si includono anche ***SI*** e ***DI***:

Registro	Definizione	Scopo prevalente
AX	accumulatore	Usato soprattutto nei calcoli e per l'input e output. Nelle convenzioni di chiamata comuni, si usa ***AX*** per restituire un valore attraverso una funzione.
BX	base	Viene usato particolarmente come indice da sommare ad altri, per individuare una posizione in memoria.
CX	contatore	Usato come contatore nei cicli.
DX	dati	Si affianca a ***AX***, soprattutto nelle divisioni e moltiplicazioni.
SI	*source index*	Usato prevalentemente come indice dell'origine, nell'ambito di un segmento dati (***DS*** o ***ES***).
DI	*destination index*	Usato prevalentemente come indice della destinazione, nell'ambito di un segmento dati (***DS*** o ***ES***).

89.1.3 Trasferimento di dati tra due segmenti differenti

«

Il trasferimento di dati tra segmenti di memoria differenti richiede l'uso di istruzioni apposite, con cui il registro ***DS*** individua il segmento di origine e ***ES*** quello di destinazione. Viene mostrato un esempio, con una porzione di codice, che ha lo scopo di copiare un intero segmento, dall'indirizzo efficace 10000_{16}, a $1FFFF_{16}$ incluso, a partire dall'indirizzo 30000_{16}, fino a $3FFFF_{16}$. La notazione è quella «Intel».

```
cld              ; Azzera l'indicatore di direzione.
mov   ax, 3000h ; Assegna a ES il segmento di destinazione,
mov   es, ax    ; attraverso AX.
mov   ax, 1000h ; Assegna a DS il segmento di origine,
mov   ds, ax    ; attraverso AX.
mov   cx, 8000h ; Imposta il contatore a 32768.
mov   si, 0h    ; Indice iniziale nel segmento di origine.
mov   di, 0h    ; Indice iniziale nel segmento di
                ; destinazione.
rep             ; Ripete l'istruzione successiva finché
                ; CX != 0; riducendo CX di una unità a ogni
                ; ciclo.
movsw           ; Copia 16 bit da DS:SI a ES:DI,
                ; incrementando di due unità sia SI, sia DI
                ; (in base all'indicatore di direzione).
```

Va osservato che ***CX*** riceve inizialmente un valore pari a metà della dimensione di un segmento, perché la copia avviene a coppie di byte, ovvero a interi di 16 bit. Si può notare anche che i registri di segmento coinvolti ricevono il valore attraverso la mediazione di ***AX***, perché non gli si può assegnare direttamente un valore immediato.

89.1.4 Riferimenti a indirizzi di memoria con i registri

Per indicare un indirizzo di memoria, generalmente si può utilizzare una costante numerica pura e semplice, ovvero un valore immediato, ma spesso è possibile combinare il valore di uno o più registri. Nella notazione Intel, per specificare che il risultato di un'espressione rappresenta un indirizzo di memoria, la si racchiude tra parentesi quadre. Per esempio, '`-2[DX+SI]`' fa riferimento all'indirizzo di memoria efficace che si ottiene come ***DS***·16+***DX***+***SI***−2 (***DS*** partecipa in quanto si fa riferimento a un segmento e può trattarsi solo di quello dei dati). Le combinazioni ammissibili sono rappresentate dal modello seguente, tenendo conto che qui le parentesi quadre indicano un blocco opzionale:

[***costante***] + [***BX*** | ***BP***] + [***SI*** | ***DI***]

Lo specchietto successivo riepiloga tutte le combinazioni ammissibili, dove la sigla ***imm*** rappresenta un valore immediato (una costante numerica letterale) che può essere sia positivo, sia negativo:

Notazione	Indirizzo efficace corrispondente	Notazione	Indirizzo efficace corrispondente
`[SI]`	$DS \cdot 16 + SI$	*imm* `[SI]`	$DS \cdot 16 + SI + imm$
`[DI]`	$DS \cdot 16 + DI$	*imm* `[DI]`	$DS \cdot 16 + DI + imm$
`[BP]`	$DS \cdot 16 + BP$	*imm* `[BP]`	$DS \cdot 16 + BP + imm$
`[BX]`	$DS \cdot 16 + BX$	*imm* `[BX]`	$DS \cdot 16 + BX + imm$
`[BX+SI]`	$DS \cdot 16 + BX + SI$	*imm* `[BX+SI]`	$DS \cdot 16 + BX + SI + imm$
`[BX+DI]`	$DS \cdot 16 + BX + DI$	*imm* `[BX+DI]`	$DS \cdot 16 + BX + DI + imm$
`[BP+SI]`	$DS \cdot 16 + BP + SI$	*imm* `[BP+SI]`	$DS \cdot 16 + BP + SI + imm$
`[BP+DI]`	$DS \cdot 16 + BP + DI$	*imm* `[BP+DI]`	$DS \cdot 16 + BP + DI + imm$

Si osservi che la costante letterale che precede il gruppo tra parentesi quadre può essere sostituita da un nome simbolico, con il quale si indica una variabile in memoria (preferibilmente un array). In tal modo, la notazione richiama quella degli array, come si fa con il linguaggio C. Per esempio, '**`x[SI]`**', individua così il byte ***SI***-esimo a partire dall'indirizzo a cui si riferisce ***x***.

89.1.5 Convenzioni di chiamata

Le convenzioni di chiamata adottate per i microprocessori x86-16 sono le stesse di quelle usate per x86-32:

- si inseriscono nella pila dei dati gli argomenti della chiamata, in ordine inverso, in modo che l'ultimo inserimento sia quello del primo parametro della funzione;

```
push …
```

- si esegue la chiamata;

```
call …
```

- all'interno della funzione si salva il valore di ***BP*** nella pila, si assegna a ***BP*** l'indice attuale della pila (in modo da poter usare ***BP*** come riferimento per raggiungere nella pila gli argomenti della chiamata e le variabili locali) e si allocano nella stessa le variabili locali (variabili automatiche);

```
enter …
```

- si salvano nella pila i registri che la funzione va a modificare, quindi si procede con il lavoro della funzione;

```
pusha
```

- il primo argomento della chiamata si raggiunge con '**+4[*BP*]**', il secondo con '**+6[*BP*]**',... la prima variabile locale si raggiunge con '**-2[*BP*]**', la seconda con '**-4[*BP*]**',...

Al termine della funzione si fa in modo di ripristinare la situazione precedente alla chiamata, restituendo eventualmente un valore attraverso il registro ***AX*** o eventualmente la coppia ***DX***:***AX***;

- vengono ripristinati i registri salvati all'inizio della funzione;

```
popa
```

- se la funzione deve restituire un valore viene, questo viene assegnato a ***AX***, oppure ***DX***:***AX*** (se questo valore è da 32 bit);

```
mov ax, -m[bp]
mov dx, -n[bp]
```

- viene ridotta la pila riportandone l'indice al valore di ***BP*** e recuperando il valore precedente di ***BP***;

```
leave
```

- si ritorna all'indirizzo successivo alla chiamata;

```
ret
```

- si espellono gli argomenti della chiamata.

```
pop …
```

89.1.6 Sintesi delle istruzioni x86-16

Nelle tabelle successive vengono annotate le istruzioni che possono essere utilizzate con i microprocessori x86-16, raggruppate secondo il contesto a cui appartengono. Sono però escluse le istruzioni '**AA*x***' e '**DA*x***', relative alla gestione dei numeri in formato BCD (*Binary coded decimal*).

L'ordine in cui sono specificati gli operandi è quello «Intel», ovvero appare prima la destinazione e poi l'origine. Le sigle usate per definire i tipi di operandi sono: ***reg*** per «registro»; ***mem*** per «memoria»; ***imm*** per «immediato» (costante numerica).

Quando appare la pseudocodifica che deve spiegare l'effetto di un'istruzione, i riferimenti agli indirizzi in memoria vengono fatti in modo inusuale. Per esempio, ***(DS·16+SI)*** indica un indirizzo in memoria, individuato dal registro ***SI*** che si riferisce al segmento annotato in ***DS***. In modo analogo, ****(DS·16+SI)*** individua il contenuto della memoria al tale indirizzo, mentre ***&nome*** rappresenta l'indirizzo in memoria del simbolo ***nome***.

Nella colonna degli indicatori appare: il simbolo «#» per annotare che l'indicatore relativo può essere modificato dall'istruzione; il simbolo «t» per annotare che lo stato precedente dell'indicatore viene considerato dall'istruzione; zero o uno se l'indicatore viene impostato in un certo modo; il simbolo «?» se l'effetto dell'istruzione sull'indicatore è indefinito.

Tabella 89.16. Assegnamenti, scambi, conversioni e istruzione nulla.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`NOP`		*not operate* Istruzione nulla.	`cpazstido` `.........`
`MOV`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	*move* Copia il valore dell'origine nella destinazione. Consente la copia da e verso i registri di segmento, ma per assegnare un valore a un registro di segmento occorre eseguire un passaggio intermedio attraverso un registro per usi generali. Origine e destinazione devono avere la stessa quantità di bit. ***dst*** := ***org***	`cpazstido` `.........`
`LEA`	***reg***, ***mem***	*load effective address* Mette nel registro l'indirizzo della memoria, inteso come scostamento dall'inizio del segmento dati. ***dst*** := &***org***	`cpazstido` `.........`

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori
LDS	***reg***, ***mem***	*load pointer using DS* Carica dalla memoria un valore a 32 bit, diviso in due blocchi da 16 bit, mettendo il primo blocco nel registro indicato e mettendo il secondo nel registro ***DS*** (segmento dati). ***DS*** : ***dst*** := ***org***	cpazstido ....
LES	***reg***, ***mem***	*load pointer using ES* Carica dalla memoria un valore a 32 bit, diviso in due blocchi da 16 bit, mettendo il primo blocco nel registro indicato e mettendo il secondo nel registro ***ES***. ***ES*** : ***dst*** := ***org***	cpazstido ....
XCHG	***reg***, ***reg*** ***reg***, ***mem*** ***mem***, ***reg***	*exchange data* Scambia i valori. ***dst*** :==: ***org***	cpazstido ....
CBW		*convert byte to word* Converte un intero con segno, della dimensione di 8 bit, contenuto in ***AL***, in modo da occupare tutto ***AX*** (da 8 bit a 16 bit). L'espansione tiene conto del segno. ***AX*** := ***AL***	cpazstido ....

CWD		*convert word to double word*
		Converte un intero con segno, della dimensione di 16 bit, contenuto in ***AX***, in modo da estendersi anche in ***DX***, tenendo conto del segno.
		IF ***AX*** >= 0
		THEN
		    DX := 0
		ELSE
		    DX := $\texttt{FFFF}_{16}$
			cpazstido
			....

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori
`LAHF`		*load flags into AH* Carica i primi otto indicatori in ***AH***, escludendo quelli riservati. $AH_{bit0}:=c$ $AH_{bit1}:=1$ $AH_{bit2}:=p$ $AH_{bit3}:=0$ $AH_{bit4}:=a$ $AH_{bit5}:=0$ $AH_{bit6}:=z$ $AH_{bit7}:=s$	`cpazstido` `#########`
`SAHF`		*store AH into flags* Modifica il valore dei primi otto indicatori, esclusi i bit 1, 3 e 5 (il secondo, il quarto e il sesto, che sono riservati e a loro non si attribuisce un significato particolare), scrivendoci sopra il contenuto di ***AH***. $c:=AH_{bit0}$ $p:=AH_{bit2}$ $a:=AH_{bit4}$ $z:=AH_{bit6}$ $s:=AH_{bit7}$	`cpazstido` `#########`

Tabella 89.17. Movimento di dati.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori

`LODSB`		*load string byte*
		Dall'indirizzo a cui punta la coppia ***DS:SI*** (***DS·16+SI***), viene letto un byte e copiato in ***AL***. Se l'indicatore di direzione è pari a zero, ***SI*** viene incrementato di una unità, altrimenti viene decrementato di una unità.
		AL :=****(DS·16+SI)***
		`IF` ***d***`==0`
		`THEN`
		    SI`++`
		`ELSE`
		    SI`--`
			`cpazstido`
			`........t.`
			`.........`

`LODSW`		*load string word*
		Dall'indirizzo a cui punta la coppia ***DS:SI*** (***DS·16+SI***), viene letto un blocco da 16 bit e copiato in ***AX***. Se l'indicatore di direzione è pari a zero, ***SI*** viene incrementato di due unità, altrimenti viene decrementato di due unità.
		AL :=****(DS·16+SI)***
		`IF` ***d***`==0`
		`THEN`
		    SI :`+=2`
		`ELSE`
		    SI :`-=2`
			`cpazstido`
			`........t.`
			`.........`

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori
STOSB		*store string byte* All'indirizzo a cui punta la coppia ***ES:DI*** (***ES·16+DI***), viene scritto il valore contenuto in ***AL***, aggiornando ***DI*** in base al contenuto dell'indicatore di direzione. ****(ES·16+DI)*** :=***AL*** IF ***d***==0 THEN ***DI***++ ELSE ***DI***−−	cpazstido ....t. ....
STOSW		*store string word* All'indirizzo a cui punta la coppia ***ES:DI*** (***ES·16+DI***), viene scritto il valore contenuto in ***AX***, aggiornando ***DI*** in base al contenuto dell'indicatore di direzione. ****(ES·16+DI)*** :=***AX*** IF ***d***==0 THEN ***DI*** :+=2 ELSE ***DI*** :−=2	cpazstido ....t. ....

Nome	Operandi: ***dst*, *org1*, *org2***	Descrizione	Indicatori

`MOVSB`		*move string byte*
		Copia un byte, dall'indirizzo a cui punta la coppia ***DS:SI*** (***DS·16+SI***), all'indirizzo a cui punta la coppia ***ES:DI*** (***ES·16+DI***), aggiornando ***SI*** e ***DI*** in base al valore dell'indicatore di direzione.
		*(ES·16+DI) := *(DS·16+SI)
		IF ***d***==0
		THEN
		    SI++
		    DI++
		ELSE
		    SI--
		    DI--
			`cpazstido`
			`.......t.`
			`.........`

`MOVSW`		*move string word*
		Copia un blocco di 16 bit, dall'indirizzo a cui punta la coppia ***DS:SI*** (***DS·16+SI***), all'indirizzo a cui punta la coppia ***ES:DI*** (***ES·16+DI***), aggiornando ***SI*** e ***DI*** in base al valore dell'indicatore di direzione.
		*(ES·16+DI) := *(DS·16+SI)
		IF ***d***==0
		THEN
		    SI :+=2
		    DI :+=2
		ELSE
		    SI :-=2
		    DI :-=2
			`cpazstido`
			`.......t.`
			`.........`

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori

REP		*repeat*
		Ripete l'istruzione successiva (che può essere una tra: '**LODSB**', '**LODSW**', '**STOSB**', '**STOSW**', '**MOVSB**', '**MOVSW**'), per ***CX*** volte.
		`IF` ***CX*** `!=0`
		`THEN`
		`istruzione successiva`
		CX `--`
		`ELSE`
		`break`
			`cpazstido`
			`.........`

XLATB		*translate table to byte* Assegna a ***AL*** il valore che si può raggiungere all'indirizzo composto da ***DS***:***BX+AL*** (***DS·16+BX+AL***), dove ***AL*** va inteso come valore senza segno. ***AL*** `:=`****(DS·16+BX+AL)***	`cpazstido` `.........`

Tabella 89.18. Confronti con la memoria.

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori

SCASB		*compare string byte*
		Confronta il contenuto di ***AL*** con il valore a cui punta la coppia ***ES:DI*** (***ES·16+DI***), aggiornando di conseguenza gli indicatori e anche il registro ***DI*** in base all'indicatore di direzione.
		****(ES·16+DI)−AL***
		`IF` ***d*** `==0`
		`THEN`
		DI `++`
		`ELSE`
		DI `--`
			`cpazstido`
			`·······t·`
			`#####···#`

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori

`SCASW`		*compare string word*
		Confronta il contenuto di ***AX*** con il valore a cui punta la coppia ***ES:DI*** (***ES·16+DI***), aggiornando di conseguenza gli indicatori e anche il registro ***DI*** in base all'indicatore di direzione.
		****(ES·16+DI)−AX***
		`IF` ***d***`==0`
		`THEN`
		    DI `:+=2`
		`ELSE`
		    DI `:-=2`
			`cpazstido`
			`·······t·`
			`#####···#`

`CMPSB`		*compare string byte in memory*
		Confronta il byte a cui punta la coppia ***ES:DI*** (***ES·16+DI***), con quello a cui punta la coppia ***DS:SI*** (***DS·16+SI***), aggiornando di conseguenza gli indicatori e anche i registri ***DI*** e ***SI*** in base all'indicatore di direzione.
		****(DS·16+SI)−*(ES·16+DI)***
		`IF` ***d***`==0`
		`THEN`
		    SI`++`
		    DI`++`
		`ELSE`
		    SI`++`
		    DI`--`
			`cpazstido`
			`·······t·`
			`#####···#`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori

`CMPSW`		*compare string word in memory*
		Confronta il blocco da 16 bit a cui punta la coppia ***ES:DI*** (***ES·16+DI***), con quello a cui punta la coppia ***DS:SI*** (***DS·16+SI***), aggiornando di conseguenza gli indicatori e anche i registri ***DI*** e ***SI*** in base all'indicatore di direzione.
		****(DS·16+SI)−*(ES·16+DI)***
		`IF` ***d***`==0`
		`THEN`
		    SI`++`
		    DI`++`
		`ELSE`
		    SI`++`
		    DI`--`
			`cpazstido`
			`·······t·`
			`#####···#`

Nome	Operandi: ***dst*, *org1*, *org2***	Descrizione	Indicatori

REPE
 	
REPZ
		repeat while equal
		repeat while zero
		Ripete l'istruzione successiva (che può essere una tra: '**SCASB**', '**SCASW**', '**CMPSB**', '**CMPSW**'), fino a che l'indicatore ***z*** è pari a uno (rappresentante l'uguaglianza di una comparazione, ovvero che la sottrazione dà zero), fino a un massimo di ***CX*** volte.
		`IF` ***CX***`!=0`
		`THEN`
		    `istruzione successiva`
		    CX`--`
		    `IF` ***z***`==1`
		    `THEN`
		        `continue`
		    `ELSE`
		        `break`
		`ELSE`
		    `break`
			`cpazstido`
			`...#.....`

Nome	Operandi: ***dst*, *org1*, *org2***	Descrizione	Indicatori

`REPNE`
 	
`REPNZ`
		repeat while not equal
		repeat while not zero
		Ripete l'istruzione successiva (che può essere una tra: '**`SCASB`**', '**`SCASW`**', '**`CMPSB`**', '**`CMPSW`**'), fino a che l'indicatore ***z*** è pari a zero (rappresentante la disuguaglianza della comparazione, ovvero che la sottrazione non dà zero), fino a un massimo di ***CX*** volte.
		`IF` ***CX***`!=0`
		`THEN`
		    `istruzione successiva`
		    CX`--`
		    `IF` ***z***`==0`
		    `THEN`
		        `continue`
		    `ELSE`
		        `break`
		`ELSE`
		    `break`
			`cpazstido`
			`···#·····`

Tabella 89.19. Operazioni aritmetiche.

Nome	Operandi: ***dst*, *org1*, *org2***	Descrizione	Indicatori
`NEG`	***reg*** ***mem***	*negation* Inverte il segno di un numero, attraverso il complemento a due. ***operand*** := −***operand***	`cpazstido` `##·##···#`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`ADD`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	*addition* Somma di interi, con o senza segno, ignorando il riporto precedente. Se i valori si intendono con segno, è importante l'esito dell'indicatore di traboccamento (*overflow*), se invece i valori sono da intendersi senza segno, è importante l'esito dell'indicatore di riporto (*carry*). ***dst*** := ***org*** + ***dst***	`cpazstido` `##·##···#`
`SUB`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	*subtraction* Sottrazione di interi con o senza segno, ignorando il riporto precedente. ***dst*** := ***org*** − ***dst***	`cpazstido` `##·##···#`

`ADC`	***reg***, ***reg***
	reg, ***mem***
	reg, ***imm***
	mem, ***reg***
	mem, ***imm***
		addition with carry
		Somma di interi, con o senza segno, aggiungendo anche il riporto precedente (l'indicatore *carry*).
		dst := ***org*** + ***dst*** + ***c***
			`cpazstido`
			`t········`
			`##·##···#`

`SBB`	***reg***, ***reg***
	reg, ***mem***
	reg, ***imm***
	mem, ***reg***
	mem, ***imm***
		subtraction with borrow
		Sottrazione di interi, con o senza segno, tenendo conto del «prestito» precedente (l'indicatore *carry*).
		dst := ***org*** + ***dst*** − ***c***
			`cpazstido`
			`t········`
			`##·##···#`

`INC`	***reg*** ***mem***	*increment* Incrementa di una unità un intero. ***operand***++	`cpazstido` `·#·##···#`
`DEC`	***reg*** ***mem***	*decrement* Decrementa di una unità un valore intero. ***operand***−−	`cpazstido` `·#·##···#`

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori
`MUL`	***reg*** ***mem***	*multiply* Moltiplicazione intera senza segno. L'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX := AL*operand*** ***DX:AX := AX*operand***	`cpazstido` `#?·??···#`
`DIV`	***reg*** ***mem***	*division* Divisione intera senza segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL := AX / operand*** ***AH := AX%operand*** ***AX := (DX:AX) / operand*** ***DX := (DX:AX)%operand***	`cpazstido` `??·??···?`
`IMUL`	***reg*** ***mem***	*signed multiply* Moltiplicazione intera con segno. In questo caso l'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX := AL*operand*** ***(DX:AX) := AX*operand***	`cpazstido` `#?·??···#`
`IDIV`	***reg*** ***mem***	*signed division* Divisione intera con segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL := AX / operand*** ***AH := AX%operand*** ***AX := (DX:AX) / operand*** ***DX := (DX:AX)%operand***	`cpazstido` `??·??···?`

Tabella 89.20. Operazioni logiche.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`NOT`	***reg*** ***mem***	NOT di tutti i bit dell'operando. ***dst*** := `NOT` ***dst***	`cpazstido` `.........`

`AND`
`OR`
`XOR`
	reg, ***reg***
	reg, ***mem***
	reg, ***imm***
	mem, ***reg***
	mem, ***imm***
		AND, OR, o XOR, tra tutti i bit dei due operandi.
		dst := ***org*** `AND` ***dst***
		dst := ***org*** `OR` ***dst***
		dst := ***org*** `XOR` ***dst***
			`cpazstido`
			`0#·##···0`

Tabella 89.21. Scorrimenti e rotazioni.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`SHL` `SHR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	*shift left* *shift right* Fa scorrere i bit, rispettivamente verso sinistra o verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`cpazstido` `#·······#`
`SAL` `SAR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	*shift arithmetically left* *shift arithmetically right* Fa scorrere i bit, rispettivamente verso sinistra o verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto), mantenendo il segno originale (logicamente '**`SAL`**' è identico a '**`SHL`**'). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`cpazstido` `#········`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori

RCL
RCR
	reg, 1
	mem, 1
	reg
	mem
		rotate left with carry
		rotate right with carry
		Ruota i bit, rispettivamente verso sinistra o verso destra, utilizzando anche l'indicatore di riporto (*carry*). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.
			`cpazstido`
			`t········`
			`#·······#`

ROL ROR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	*rotate left* *rotate right* Ruota i bit, rispettivamente verso sinistra o verso destra. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`cpazstido` `#·······#`

Tabella 89.22. Chiamate e gestione della pila.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
CALL	***reg*** ***mem*** ***imm***	Inserisce nella pila l'indirizzo dell'istruzione successiva e salta all'indirizzo indicato, che si riferisce allo scostamento a partire dall'inizio del segmento codice (***CS***). Pertanto, l'indirizzo a cui ci si riferisce è a 16 bit. `push` ***indirizzo_successivo*** ***IP*** := ***operand***	`cpazstido` `·········`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
CALL FAR	***imm***:***imm***	*call procedure* Inserisce nella pila il valore di ***CS*** e poi l'indirizzo dell'istruzione successiva (***IP*** dell'istruzione successiva) e salta all'indirizzo indicato. L'indirizzo deve essere di quattro byte (32 bit), in quanto deve specificare anche il segmento codice da raggiungere. push ***CS*** push ***indirizzo_successivo*** ***CS*** : ***IP*** := ***operand***	cpazstido ....
RET		*return from call* Estrae dalla pila l'indirizzo dell'istruzione da raggiungere (***IP***) e salta a quella (serve a concludere una chiamata eseguita con '**CALL**'). pop ***IP***	cpazstido ....
RETF RET FAR		*return from far call* Estrae dalla pila il valore di ***CS*** e quindi l'indirizzo dell'istruzione da raggiungere (***IP***) e salta a quella (serve a concludere una chiamata eseguita con '**CALL FAR**'). pop ***IP*** pop ***CS***	cpazstido ....
PUSH	***reg*** ***mem***	*push data onto stack* Inserisce nella pila il valore (della dimensione di un registro comune). ***SP*** :-=2 *(***SS***·16+***SP***) :=***operand***	cpazstido ....

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori
POP	***reg*** ***mem***	*pop data from stack* Estrae dalla pila l'ultimo valore inserito (della dimensione di un registro comune). ***operand*** := * (***SS*** · 16+***SP***) ***SP*** : +=2	cpazstido ....
PUSHF		*push flags onto stack* Inserisce nella pila l'insieme del registro degli indicatori (***FLAGS***). push ***FLAGS***	cpazstido ....
POPF		*pop flags from stack* Estrae dalla pila l'insieme del registro degli indicatori (***FLAGS***), aggiornando di conseguenza il registro stesso. pop ***FLAGS***	cpazstido ?????????
ENTER	***imm8***, 0	*enter stack frame* Questa funzione esiste a partire dai microprocessori i186. Inserisce nella pila il valore di ***BP***, poi assegna a ***BP*** il valore di ***SP*** e infine decrementa ***SP*** del valore fornito come immediato. Serve a predisporre ***BP*** e ***SP*** all'inizio di una funzione, specificando lo spazio necessario per le variabili locali nella pila. push ***BP*** ***BP*** : =***SP*** ***SP*** : -=2 * ***dst***	cpazstido ....

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
PUSHA		*push all registers onto stack* Questa funzione esiste a partire dai microprocessori i186. Inserisce nella pila i registri principali: ***AX***, ***CX***, ***DX***, ***BX***, ***SP***, ***BP***, ***SI***, ***DI***. push ***AX*** push ***CX*** push ***DX*** push ***BX*** push ***SP*** push ***BP*** push ***SI*** push ***DI***	cpazstido ....
POPA		*pop all registers from stack* Questa funzione esiste a partire dai microprocessori i186. Ripristina i registri principali, estraendo i contenuti dalla pila: ***DI***, ***SI***, ***BP***, ***SP*** viene eliminato senza aggiornare il registro, ***BX***, ***DX***, ***CX***, ***AX***. Come si vede, anche se '**PUSHA**' salva l'indice della pila, in pratica questo indice non viene ripristinato. pop ***DI*** pop ***SI*** pop ***BP*** ***SP*** **:** +=2 pop ***BX*** pop ***DX*** pop ***CX*** pop ***AX***	cpazstido ....

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`LEAVE`		*leave stack frame* Questa funzione esiste a partire dai microprocessori i186. Ripristina i valori di ***BP*** e di ***SP***, allo stato che avevano prima dell'uso dell'istruzione '**`ENTER`**'. ***SP***:=***BP*** `pop` ***BP***	`cpazstido` `.........`

Tabella 89.23. Interruzioni.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori

`INT`	***imm8***	*call to interrupt*
		Esegue una chiamata attraverso un'interruzione.
		Prima di saltare alla codice relativo all'interruzione selezionata, inserisce nella pila ***FLAGS***, ***CS*** e ***IP***. Azzera anche l'indicatore ***IF*** (*interrupt flag*), mentre gli altri indicatori rimangono inalterati.
		`pusf`
		`push` ***CS***
		`push` ***IP***
		i:=`0`
		`jmp far 0:(`***operand***`·4)`
			`cpazstido`
			`.........`

`IRET`		*return from interrupt* Conclude l'esecuzione del codice relativo a un'interruzione recuperando dalla pila i valori inseriti alla chiamata con '**`INT`**': ***IP***, ***CS*** e ***FLAGS***. Pertanto, il valore degli indicatori viene ripristinato allo stato precedente alla chiamata. `pop` ***IP*** `pop` ***CS*** `popf`	`cpazstido` `##·##···#`

Nome	Operandi: ***dst*, *org1*, *org2***	Descrizione	Indicatori
`CLI`		*clear interrupt flag* Azzera l'indicatore di abilitazione delle interruzioni (*interrupt flag*), disabilitando di conseguenza le interruzioni hardware. ***i*** `:=0`	`cpazstido` `······0··`
`STI`		*set interrupt flag* Attiva l'indicatore di abilitazione delle interruzioni (*interrupt flag*), abilitando di conseguenza le interruzioni hardware. ***i*** `:= 1`	`cpazstido` `······1··`
`HLT`		*enter halt state* Ferma il sistema, fino a quando viene ricevuta un'interruzione hardware.	`cpazstido` `·········`
`INTO`		*interrupt if overflow* Se l'indicatore di straripamento è attivo, esegue la chiamata dell'interruzione numero 4 (la quale dovrebbe gestore il problema).	`cpazstido` `········t` `·········`

Tabella 89.24. Indicatori e confronti tra registri.

Nome	Operandi: ***dst*, *org1*, *org2***	Descrizione	Indicatori
`CLC`		*clear carry flag* Azzera l'indicatore del riporto (*carry*), senza intervenire negli altri indicatori. ***c*** `:=0`	`cpazstido` `0········`
`CLD`		*clear direction flag* Azzera l'indicatore di direzione (*direction*), senza intervenire negli altri indicatori. ***d*** `:=0`	`cpazstido` `·······0·`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`STC`		*set carry flag* Attiva l'indicatore di riporto (*carry*). ***c*** **:**=`1`	`cpazstido` `1········`
`STD`		*set direction flag* Attiva l'indicatore di direzione (*direction*). ***d*** **:**=`1`	`cpazstido` `·······1·`
`CMC`		*complement carry flag* Inverte il valore dell'indicatore del riporto (*carry*).	`cpazstido` `#········`
`CMP`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	*compare operands* Confronta due valori interi. La comparazione avviene simulando la sottrazione dell'origine dalla destinazione, senza però modificare gli operandi, ma aggiornando gli indicatori, come se fosse avvenuta una sottrazione vera e propria. ***dst*** − ***org***	`cpazstido` `##·##···#`
`TEST`	***reg***, ***reg*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	*logical compare* AND dei due valori senza conservare il risultato. Serve solo a ottenere l'aggiornamento degli indicatori. ***dst*** `AND` ***org***	`cpazstido` `0#·##···0`

Tabella 89.25. Salti.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`JMP`	***reg*** ***mem*** ***imm***	*jump* Salto incondizionato all'indirizzo indicato, che si intende relativo al segmento codice (***CS***). ***IP* :=*operand***	`cpazstido` `·········`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`JMP FAR`	***imm***:***imm***	*far jump* Salto incondizionato all'indirizzo indicato, costituito sia dal segmento codice, sia dall'indirizzo relativo, all'interno di questo. ***CS*** : ***IP*** : =***operand***	`cpazstido` `·········`
`JA` `JNBE`	***imm***	*conditional jump* Dopo un confronto di valori senza segno, salta se la destinazione era maggiore dell'origine. Il salto riguarda solo l'ambito del segmento codice attuale. `CMP` ***dst***, ***org*** `IF` ***dst*** `>` ***org*** `THEN` `go to` ***imm***	`cpazstido` `t··t·····`
`JAE` `JNB`	***imm***	*conditional jump* Dopo un confronto di valori senza segno, salta se la destinazione era maggiore o uguale all'origine. Il salto riguarda solo l'ambito del segmento codice attuale. `CMP` ***dst***, ***org*** `IF` ***dst*** `>=` ***org*** `THEN` `go to` ***imm***	`cpazstido` `t··t·····`

Nome	Operandi: *dst*, *org1*, *org2*	Descrizione	Indicatori

`JB`
 	
`JNAE`
	imm	*conditional jump*
		Dopo un confronto di valori senza segno, salta se la destinazione era minore dell'origine. Il salto riguarda solo l'ambito del segmento codice attuale.
		`CMP` ***dst*** **,** ***org***
		`IF` ***dst*** `<` ***org***
		`THEN`
		`go to` ***imm***
			`cpazstido`
			`t··t·····`

`JBE`
 	
`JNA`
	imm	*conditional jump*
		Dopo un confronto di valori senza segno, salta se la destinazione era minore o uguale all'origine. Il salto riguarda solo l'ambito del segmento codice attuale.
		`CMP` ***dst*** **,** ***org***
		`IF` ***dst*** `<=` ***org***
		`THEN`
		`go to` ***imm***
			`cpazstido`
			`t··t·····`

`JE`	***imm***	*conditional jump* Dopo un confronto, indipendentemente dal segno, salta se la destinazione era uguale all'origine. Il salto riguarda solo l'ambito del segmento codice attuale. `CMP` ***dst*** **,** ***org*** `IF` ***dst*** `==` ***org*** `THEN` `go to` ***imm***	`cpazstido` `···t·····`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
JNE	***imm***	*conditional jump* Dopo un confronto, indipendentemente dal segno, salta se la destinazione era diversa dall'origine. Il salto riguarda solo l'ambito del segmento codice attuale. CMP ***dst***, ***org*** IF ***dst*** != ***org*** THEN go to ***imm***	cpazstido ···t·····
JG JNLE	***imm***	*conditional jump* Dopo un confronto con segno, salta se la destinazione era maggiore dell'origine. Il salto riguarda solo l'ambito del segmento codice attuale. CMP ***dst***, ***org*** IF ***dst*** > ***org*** THEN go to ***imm***	cpazstido ···tt···t
JGE JNL	***imm***	*conditional jump* Dopo un confronto con segno, salta se la destinazione era maggiore o uguale all'origine. Il salto riguarda solo l'ambito del segmento codice attuale. CMP ***dst***, ***org*** IF ***dst*** >= ***org*** THEN go to ***imm***	cpazstido ···tt···t

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`JL` `JNGE`	***imm***	*conditional jump* Dopo un confronto con segno, salta se la destinazione era minore dell'origine. Il salto riguarda solo l'ambito del segmento codice attuale. `CMP` ***dst*** `,` ***org*** `IF` ***dst*** `<` ***org*** `THEN` `go to` ***imm***	`cpazstido` `···tt···t`
`JLE` `JNG`	***imm***	*conditional jump* Dopo un confronto con segno, salta se la destinazione era minore o uguale all'origine. Il salto riguarda solo l'ambito del segmento codice attuale. `CMP` ***dst*** `,` ***org*** `IF` ***dst*** `<=` ***org*** `THEN` `go to` ***imm***	`cpazstido` `···tt···t`
`JC` `JNC`	***imm***	*conditional jump* Salta se l'indicatore del riporto (*carry*), rispettivamente, è attivo, oppure non è attivo. Il salto riguarda solo l'ambito del segmento codice attuale.	`cpazstido` `t········`
`JO` `JNO`	***imm***	*conditional jump* Salta se l'indicatore di traboccamento (*overflow*), rispettivamente, è attivo, oppure non è attivo.	`cpazstido` `········t`
`JS` `JNS`	***imm***	*conditional jump* Salta se l'indicatore di segno (*sign*), rispettivamente, è attivo, oppure non è attivo.	`cpazstido` `····t····`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`JZ` `JNZ`	***imm***	*conditional jump* Salta se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.	`cpazstido` `···t·····`
`JP` `JPE`	***imm***	*conditional jump* Salta se l'indicatore di parità è attivo.	`cpazstido` `·t·······`
`JNP` `JPO`	***imm***	*conditional jump* Salta se l'indicatore di parità non è attivo.	`cpazstido` `·t·······`
`JCXZ`	***imm***	*conditional jump* Salta se il valore contenuto nel registro ***CX*** è pari a zero.	`cpazstido` `·········`

Tabella 89.26. Iterazioni

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori
`LOOP`	***imm8***	*loop* Senza alterare gli indicatori, decrementa di una unità il registro '**`CX`**', quindi, se il registro è ancora diverso da zero, salta all'indirizzo cui fa riferimento l'operando. Tale indirizzo non può essere molto lontano dalla posizione corrente.	`cpazstido` `·········`
`LOOPE` `LOOPZ`	***imm8***	*conditional loop* Senza alterare gli indicatori, decrementa di una unità il registro '**`CX`**', quindi, se il registro è ancora diverso da zero e l'indicatore «zero» è attivo, salta all'indirizzo cui fa riferimento l'operando. Tale indirizzo non può essere molto lontano dalla posizione corrente.	`cpazstido` `···t·····`

Nome	Operandi: ***dst*, *org1*, *org2***	Descrizione	Indicatori
LOOPNE LOOPNZ	***imm8***	*conditional loop* Senza alterare gli indicatori, decrementa di una unità il registro '**CX**', quindi, se il registro è ancora diverso da zero e l'indicatore «zero» non è attivo, salta all'indirizzo cui fa riferimento l'operando. Tale indirizzo non può essere molto lontano dalla posizione corrente.	`cpazstido` `···t·····`

Tabella 89.27. Input e output.

Nome	Operandi: ***dst*, *org1*, *org2***	Descrizione	Indicatori
IN	***AL*, *imm8*** ***AX*, *imm8***	*input* Assegna a ***AL*** o ***AX*** il valore letto dalla porta specificata; in tal caso il numero di porta non può essere superiore a 255.	`cpazstido` `·········`
IN	***AL*, *DX*** ***AX*, *DX***	*input* Assegna a ***AL*** o ***AX*** il valore letto dalla porta specificata da ***DX***; in tal caso il numero di porta può essere superiore a 255.	`cpazstido` `·········`
OUT	***imm8*, *AL*** ***imm8*, *AX***	*output* Scrive nella porta specificata il valore contenuto in ***AL*** o ***AX***; in tal caso il numero di porta non può essere superiore a 255.	`cpazstido` `·········`
OUT	***DX*, *AL*** ***DX*, *AX***	*output* Scrive nella porta indicata da ***DX*** il valore contenuto in ***AL*** o ***AX***; in tal caso il numero di porta può essere superiore a 255.	`cpazstido` `·········`

89.1.7 Sostituzione delle istruzioni per i186

«

Nella sezione precedente sono state menzionate delle istruzioni che non fanno parte dei microprocessori 8086/8088, ma queste possono essere ottenute facilmente attraverso altre istruzioni elementari, tanto che l'assemblatore potrebbe provvedervi direttamente. A ogni modo viene annotato qui come possono essere sostituite.

Listato 89.28. Sostituzione per l'istruzione '**PUSHA**'.

```
push ax
push cx
push dx
push bx
push sp
push bp
push si
push di
```

Listato 89.29. Sostituzione per l'istruzione '**POPA**'. Il registro ***SP*** non viene ripristinato, di conseguenza si riduce l'indice della pila (si incrementa ***SP***) senza estrarne il valore.

```
pop di
pop si
pop bp
add sp, 2       ; non ripristina SP
pop bx
pop dx
pop cx
pop ax
```

Listato 89.30. Sostituzione per l'istruzione '**ENTER**'. La riduzione di ***SP*** dipende dalla quantità di variabili locali che si vogliono gestire. Usando interi da 16 bit, si tratta di moltiplicare la quantità di variabili locali per due. Va ricordato che il segmento a cui si riferisce ***BP*** è ***DS***, per cui è indispensabile che ***DS*** sia uguale a ***SS***, essendo usato in questo modo come riferimento alla pila.

```
push bp
mov  bp, sp
sub  sp, 2      ; 0, 2, 4, 6,...
```

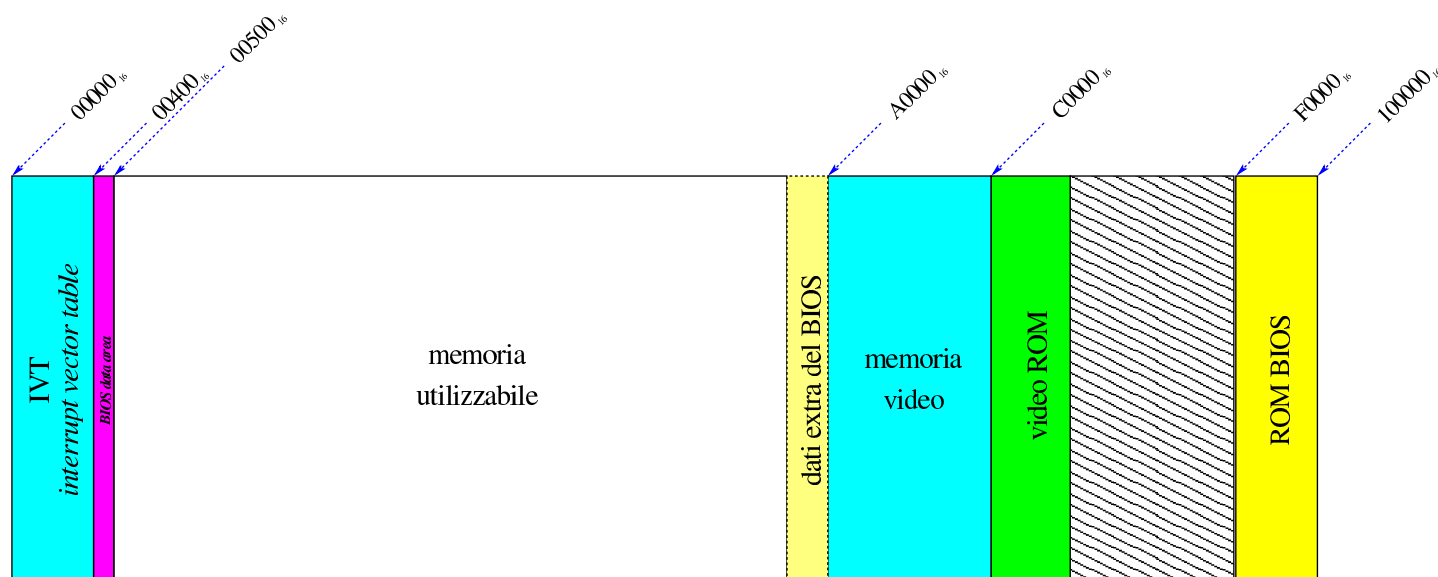
Listato 89.31. Sostituzione per l'istruzione '**LEAVE**'.

```
mov sp, bp
pop bp
```

89.2 Architettura IBM PC

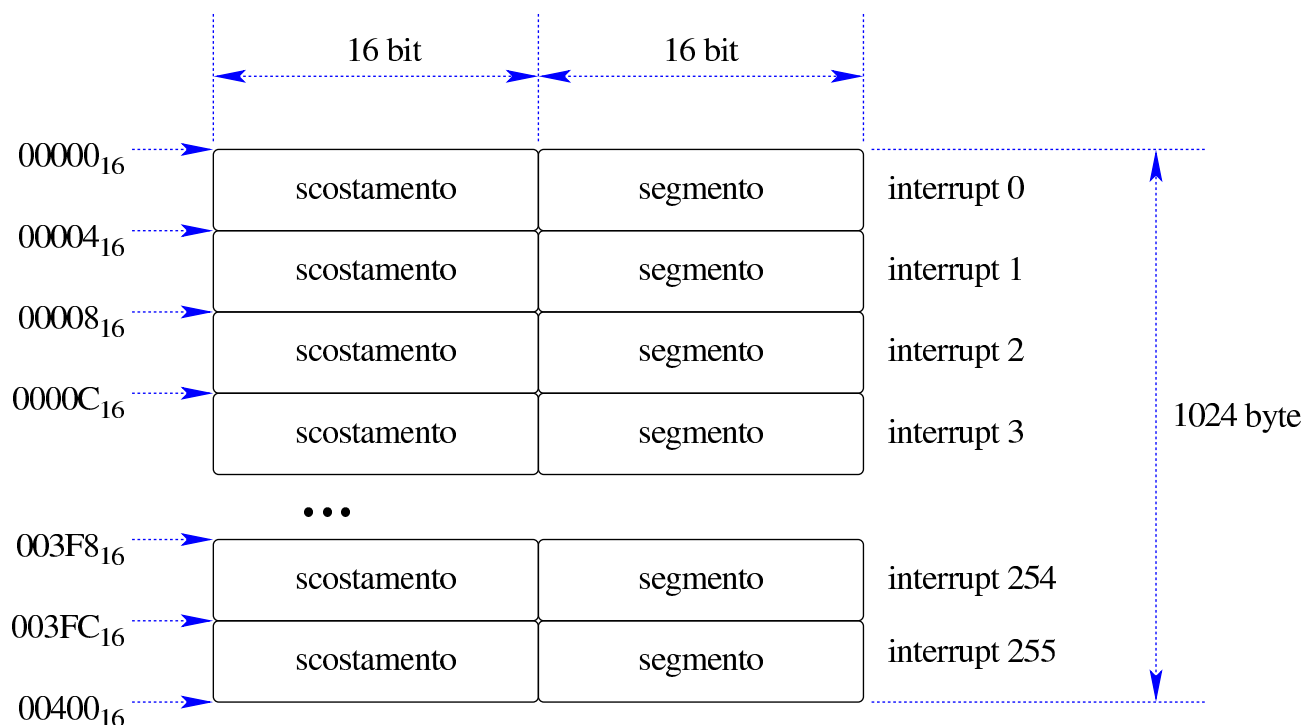
L'architettura del vecchio IBM PC prevede 1 Mibyte di memoria, in cui alcune fasce hanno degli scopi particolari e non possono essere utilizzate diversamente. Quando si programma a 32 bit, di norma si dispone, in proporzione, di una quantità enorme di memoria, per cui di solito lo spazio inferiore a 1 Mibyte viene semplicemente ignorato e si considera solo lo spazio successivo; ma se la programmazione avviene a 16 bit, si deve operare in quello spazio ristretto.

Buona parte della memoria di un elaboratore conforme all'architettura del IBM PC è impegnata per il codice del BIOS, in cui sono contenute diverse routine attivate da interruzioni, hardware o software. Nella limitazione dell'architettura originale, il sistema offerto di gestione delle interruzioni consentirebbe un utilizzo uniforme dell'hardware.

Figura 89.32. Mappa generale della memoria.

89.2.1 IVT: «interrupt vector table»

I microprocessori x86-16 utilizzano una tabella delle interruzioni collocata nella prima parte della memoria, a partire dall'indirizzo 0. La tabella è nota con la sigla IVT, ovvero *Interrupt vector table*.

Le voci della tabella IVT sono costituite semplicemente da due informazioni: l'indirizzo di segmento e lo scostamento in cui l'interruzione rispettiva viene gestita, attraverso del codice appropriato. Va osservato però che appare prima lo scostamento e poi il segmento.

Figura 89.33. Tabella IVT in memoria.

Supponendo sia ***n*** il numero di un'interruzione, si ottiene l'indirizzo della routine di interruzione con un calcolo molto semplice: lo scostamento è disponibile all'indirizzo ***n***×4; il segmento è disponibile all'indirizzo ***n***×4+2.

Solo una porzione delle voci della tabella è utilizzata in modo predefinito, mentre le altre sono disponibili per interruzioni aggiuntive a uso del sistema operativo. In ogni caso, tutta la tabella può essere modificata, in modo da dirigere le interruzioni a codice alternativo.

Tabella 89.34. Configurazione iniziale della tabella IVT, secondo l'architettura IBM PC.

Interruzione	Descrizione
00_{16}	Eccezione del microprocessore: *divide error*.
01_{16}	Eccezione del microprocessore: *debug*.
02_{16}	Eccezione del microprocessore: *non maskable interrupt*.
03_{16}	Eccezione del microprocessore: *breakpoint*.
04_{16}	Eccezione del microprocessore: *arithmetic overflow*.
05_{16}	BIOS: il tasto [*Stampa*] o [*Print_Screen*] è stato premuto.
06_{16}	Eccezione del microprocessore: *invalid opcode*.
07_{16}	Eccezione del microprocessore: *no coprocessor*.
08_{16}	Interruzione hardware: IRQ 0, temporizzatore.
09_{16}	Interruzione hardware: IRQ 1, tastiera.
$0A_{16}$	Interruzione hardware: IRQ 2.
$0B_{16}$	Interruzione hardware: IRQ 3.
$0C_{16}$	Interruzione hardware: IRQ 4.
$0D_{16}$	Interruzione hardware: IRQ 5.

Interruzione	Descrizione
$0E_{16}$	Interruzione hardware: IRQ 6.
$0F_{16}$	Interruzione hardware: IRQ 7.
10_{16}	BIOS: video.
11_{16}	BIOS: controllo dell'apparecchiatura.
12_{16}	BIOS: memoria disponibile.
13_{16}	BIOS: unità a disco.
14_{16}	BIOS: porte seriali.
15_{16}	BIOS: funzioni varie.
16_{16}	BIOS: tastiera.
17_{16}	BIOS: porte parallele.
18_{16}	BIOS: interprete BASIC in ROM.
19_{16}	BIOS: riavvio.
$1A_{16}$	BIOS: orologio.
$1B_{16}$	BIOS: tasto [*Interr*] o [*Break*].
$1C_{16}$	BIOS: temporizzatore.
$1D_{16}$	BIOS: inizializzazione video.
$1E_{16}$	BIOS: inizializzazione gestione dischi.
$1F_{16}$	BIOS: caratteri del video.

Quando si costruisce una procedura da associare a una voce della tabella delle interruzioni, occorre considerare che prima che la procedura stessa sia raggiunta, il microprocessore inserisce nella pila delle informazioni. Nello specchietto successivo si mettono a confronto le istruzioni relative alle chiamate di interruzioni e di procedure comuni:

`int`	`call far`	`call`

`pushf`
`push cs`
`push ip`
	`push cs`
	`push ip`
		`push ip`

`iret`	`retf`	`ret`

`pop ip`
`pop cs`
`popf`
	`pop ip`
	`pop cs`
		`pop ip`

89.2.2 BIOS data area

Dopo la tabella IVT, a partire dall'indirizzo 00400_{16} e fino a $004FF_{16}$ incluso, si trova un'area di memoria utilizzata dal BIOS, per annotarvi delle informazioni. Alla fine del capitolo sono riportati i riferimenti alla documentazione che consente di interpretare il contenuto di questa area, ma quello che conta sapere è che non ci si deve scrivere, a meno di impedire alle funzioni del BIOS di operare correttamente.

89.2.3 Altre aree di memoria

A partire dall'indirizzo $A0000_{16}$, fino a $BFFFF_{16}$ incluso, si trova la memoria usata per rappresentare i dati sullo schermo. Successivamente ci sono altre aree di memoria in sola lettura (un'area precedente all'indirizzo $F0000_{16}$ potrebbe essere priva di qualunque cosa) in particolare il blocco da $F0000_{16}$ a $FFFFF_{16}$ che contiene le procedure del BIOS.

In pratica, a parte la possibilità di scrivere direttamente nella memoria video, per ottenere la rappresentazione del testo sullo schermo, la memoria da $A0000_{16}$ fino alla fine, non può essere utilizzata, ma rimane incerta una porzione di memoria antecedente l'indirizzo $A0000_{16}$ che potrebbe essere utilizzata anch'essa dalle procedure che compongono il BIOS.

Per conoscere l'ammontare di memoria libera si può leggere il valore contenuto all'indirizzo 00413_{16}, nell'ambito della *BIOS data area*, tenendo conto che si tratta di un numero a 16 bit. Quel valore indica la quantità di memoria utile, espressa in multipli di 1024 byte, ma occorre considerare che si può utilizzare solo a partire dall'indirizzo 00500_{16}, ovvero dalla fine della BDA. In alternativa, si può chiamare l'interruzione 12_{16}, ottenendo dal registro ***AX*** tale valore.

89.2.4 Interruzioni principali

Il BIOS di un elaboratore IBM PC offre una serie di funzionalità, attraverso delle interruzioni, le quali possono essere utilizzate in mancanza di funzioni più sofisticate del sistema operativo. Il testo di riferimento per le interruzioni, del BIOS e dei sistemi operativi che le estendono, è quello di Ralf Brown, annotato alla fine del capitolo. Nelle sezioni successive vengono descritte solo alcune interruzioni offerte esclusivamente da BIOS standard.

89.2.4.1 INT 10_{16}/AH=00_{16} «set video mode»

Definisce la modalità di funzionamento del video. A seconda della modalità scelta, si possono usare una o più «pagine» distinte.

Parametro	Descrizione
AH	00_{16}
AL	Modalità video: 00_{16} testo, 40×25 caratteri, 16 colori, 8 pagine; 03_{16} testo, 80×25 caratteri, 16 colori, 4 pagine.

89.2.4.2 INT 10_{16}/AH=05_{16} «select active display page»

Seleziona la pagina video attiva. La numerazione parte da zero e la quantità di pagine disponibili dipende dalla modalità scelta.

Parametro	Descrizione
AH	05_{16}
BH	Pagina video da selezionare.

89.2.4.3 INT 10_{16}/AH=$0E_{16}$ «teletype output»

Mostra un carattere sullo schermo, alla pagina specificata, facendo avanzare il cursore e facendo scorrere il testo precedente se necessario. In questa modalità di visualizzazione, i caratteri con funzioni speciali vengono interpretati secondo la tradizione, tenendo conto che <*CR*> riporta il cursore all'inizio della stessa riga e che <*LF*> fa avanzare alla riga successiva, ma senza riportare il cursore all'inizio.

Parametro	Descrizione
AH	$0E_{16}$
AL	Carattere da rappresentare.
BL	Numero della pagina video.

89.2.4.4 INT 12_{16} «get memory size»

Restituisce la dimensione della memoria utilizzabile, partendo dall'indirizzo 00000_{16}, espressa in multipli di 1024 byte. La memoria utilizzabile effettivamente inizia solo a partire dall'indirizzo 00500_{16}. Eventualmente lo stesso valore sarebbe accessibile all'indirizzo 00413_{16}, leggendo un numero da 16 bit.

Valore restituito	Descrizione
AX	Dimensione della memoria disponibile.

89.2.4.5 INT 13_{16}/AH=00_{16} «reset disk system»

Azzera il sistema di gestione dei dischi, per l'unità indicata. L'unità è un numero da zero in su per i dischetti, mentre per i dischi fissi si parte da 80_{16} in su.

Parametro	Descrizione
AH	00_{16}
DL	Numero dell'unità da azzerare.

Valore restituito	Descrizione
AH	Stato: zero indica un risultato soddisfacente, altrimenti si tratta di un errore.
c (*carry*)	Zero se ***AH*** è pari a zero; altrimenti, in presenza di un errore, l'indicatore viene attivato.

89.2.4.6 INT 13_{16}/AH=02_{16} «read disk sectors into memory»

Legge uno o più settori dal disco alla memoria. Il numero del cilindro in cui si trova il settore iniziale viene indicato utilizzando il registro ***CH*** per gli otto bit meno significativi, mentre si aggiungono altri due bit, più significativi, dal registro ***CL***.

L'unità è un numero da zero in su per i dischetti, mentre per i dischi fissi si parte da 80_{16} in su.

Parametro	Descrizione
AH	02_{16}
DL	Numero dell'unità da cui leggere.
AL	Quantità di settori da leggere (deve essere maggiore di zero).
CH, $\boldsymbol{CL}_{bit6}$, $\boldsymbol{CL}_{bit7}$	Numero del cilindro del primo settore da leggere, costituito dai bit 7 e 6 del registro ***DL*** e dal registro ***CL*** (da ***0*** a 1023).
$\boldsymbol{CL}_{bit0}$..$\boldsymbol{CL}_{bit5}$	Numero del settore, relativo alla traccia, intesa come combinazione di cilindro e testina (da 1 a 63).
DH	Numero della testina (da 0 a 63, perché i due bit più significativi potrebbero essere attribuiti ad altre funzioni).
ES:***BX*** (***ES*16+BX***)	Puntatore all'area di memoria che deve ricevere i settori letti.

Valore restituito	Descrizione
AH	Stato: zero indica un risultato soddisfacente, altrimenti si tratta di un errore.
c (*carry*)	Zero se ***AH*** è pari a zero; altrimenti, in presenza di un errore, l'indicatore viene attivato.
AL	In presenza di un errore, riporta la quantità di settori letti correttamente.

89.2.4.7 INT 13_{16}/AH=03_{16} «write disk sectors»

«

Scrive uno o più settori dalla memoria nel disco. Il numero del cilindro in cui si trova il settore iniziale viene indicato utilizzando il registro ***CH*** per gli otto bit meno significativi, mentre si aggiungono altri due bit, più significativi, dal registro ***CL***.

L'unità è un numero da zero in su per i dischetti, mentre per i dischi fissi si parte da 80_{16} in su.

Parametro	Descrizione
AH	03_{16}
DL	Numero dell'unità in cui scrivere.
AL	Quantità di settori da scrivere (deve essere maggiore di zero).
CH, CL_{bit6}, CL_{bit7}	Numero del cilindro del primo settore da scrivere, costituito dai bit 7 e 6 del registro ***DL*** e dal registro ***CL*** (da ***0*** a 1023).
CL_{bit0}..CL_{bit5}	Numero del settore, relativo alla traccia, intesa come combinazione di cilindro e testina (da 1 a 63).
DH	Numero della testina (da 0 a 63, perché i due bit più significativi potrebbero essere attribuiti ad altre funzioni).
ES:***BX*** (***ES*16+BX***)	Puntatore all'area di memoria da cui trarre i dati per la scrittura dei settori.

Valore restituito	Descrizione
AH	Stato: zero indica un risultato soddisfacente, altrimenti si tratta di un errore.
c (*carry*)	Zero se ***AH*** è pari a zero; altrimenti, in presenza di un errore, l'indicatore viene attivato.
AL	In presenza di un errore, riporta la quantità di settori scritti correttamente.

89.2.4.8 INT 16_{16}/AH=00_{16} «get keystroke from keyboard»

«

Legge un valore inserito dalla tastiera, eliminandolo dalla memoria tampone associata.

Parametro	Descrizione
AH	00_{16}

Valore restituito	Descrizione
AH	Il valore letto, secondo il codice usato dal BIOS.
AL	Il valore letto, tradotto in un carattere ASCII.

89.2.4.9 INT 16_{16}/AH=01_{16} «check for keystroke in the keyboard buffer»

Legge un valore inserito dalla tastiera, ma senza eliminarlo dalla memoria tampone associata.

Parametro	Descrizione
AH	01_{16}

Valore restituito	Descrizione
z (*zero*)	Zero se la lettura è avvenuta con successo; altrimenti, se la memoria tampone è vuota, l'indicatore risulta attivato.
AH	Il valore letto, secondo il codice usato dal BIOS.
AL	Il valore letto, tradotto in un carattere ASCII.

89.3 Strumenti di sviluppo e di utilizzo

Per chi si avvale di un sistema operativo GNU, gli strumenti per sviluppare codice per x86-16 sono costituiti dalla raccolta nota con il nome Dev86, la quale mette assieme il compilatore C Bcc,[1] l'assemblatore As86 e il «collegatore» Ld86, oltre a una libreria C adatta per produrre applicazioni per ELKS (*Embeddable Linux kernel subset*).

Considerato che strumenti del genere sono utili, presumibilmente, per realizzare un programma autonomo (*stand alone*) o il kernel di un sistema operativo, un programma di avvio facilita molto il lavoro e consente di concentrare l'attenzione su ciò che si vuole realizzare veramente. Per questo motivo, nel capitolo viene anche preso in considerazione Bootblocks per l'avvio di un sistema operativo da dischetti con file system Minix 1.

89.3.1 Preparazione

In una distribuzione GNU/Linux Debian sono disponibili i pacchetti '**`bcc`**', '**`bin86`**' e '**`elks-libc`**' che forniscono il necessario per la compilazione, ma in un altro sistema GNU può essere necessario procurarsi il pacchetto sorgente Dev86, dal quale si ottiene ciò che serve.

Se si è costretti a partire dai sorgenti di Dev86, una volta scaricato il pacchetto, questo può essere espanso in una directory qualunque nell'elaboratore GNU, come mostrato dall'esempio seguente, dove però, successivamente, si possano acquisire i privilegi dell'utente '**`root`**':

`$` **`tar xzvf Dev86src-0.16.17.tar.gz`** [*Invio*]

Si ottiene la directory '`dev86-0.16.17/`' che si articola ulteriormente. Terminata l'installazione occorre compilare questi sorgenti e installarli. In questo caso si prevede di installare Dev86 a partire da '`/opt/dev86/`':

$ **cd dev86-0.16.17** [*Invio*]

$ **make PREFIX=/opt/dev86/** [*Invio*]

Viene richiesto di intervenire su alcuni indicatori (*flag*); in generale dovrebbe andare bene ciò che viene proposto in modo predefinito:

```
 1) (ON)    Library of bcc helper functions
 2) (ON)    Minimal syscalls for BIOS level
 3) (ON)    Unix error functions
 4) (ON)    Management for /etc/passwd /etc/group /etc/utmp
 5) (OFF)   Linux-i386 system call routines GCC
 6) (ON)    GNU termcap routines
 7) (ON)    Bcc 386 floating point
 8) (ON)    Linux-i386 system call routines
 9) (ON)    Example kernel include files and syscall.dat
10) (ON)    Malloc routines
11) (ON)    Various unix lib functions
12) (ON)    Msdos system calls
13) (ON)    Regular expression lib
14) (ON)    Stdio package
15) (ON)    String and memory manipulation
16) (ON)    Linux-8086 system call routines
17) (ON)    Termios functions
18) (ON)    Unix time manipulation functions.
```

Select config option to flip [or quit] > **quit** [*Invio*]

Al termine della compilazione si passa all'installazione, cominciando dalla creazione della directory '`/opt/dev86/`'. Per fare questo occorrono i privilegi dell'utente '**`root`**':

$ **su** [*Invio*]

...

mkdir -p /opt/dev86 [*Invio*]

make install [*Invio*]

89.3.2 Bcc

Bcc[2] è un compilatore C tradizionale, ovvero fatto per la vecchia sintassi, nota con la sigla K&R. Tuttavia, con l'ausilio di un programma esterno (di norma si tratta di Unproto di Wietse Venema, incluso nella distribuzione Dev86), può compilare sorgenti scritti nella forma di un C standard, pur non potendo disporre di tutte le funzionalità di un compilatore aggiornato.

```
bcc [opzioni] file_c...
```

Tabella 89.52. Alcune opzioni per l'uso di Bcc.

Opzione	Descrizione
`-ansi`	Si avvale di un programma esterno per poter accettare un sorgente scritto secondo le convenzioni attuali del linguaggio C, pur nei limiti di quanto Bcc può poi elaborare.
`-0`	Produce un codice adatto per CPU 8086/8088.
`-S`	Produce un file in linguaggio assemblatore, da usare poi con l'assemblatore As86. In mancanza dell'opzione '**-S**' o '**-c**', si ottiene direttamente un file eseguibile, con l'intervento automatico di As86 e di Ld86.
`-c`	Produce un file oggetto, da utilizzare poi con il collegatore Ld86. In mancanza dell'opzione '**-S**' o '**-c**', si ottiene direttamente un file eseguibile, con l'intervento automatico di As86 e di Ld86.
`-o` ***nome***	Produce un file con il nome specificato.
`-I`	Non utilizza i percorsi predefiniti per l'inclusione dei file di intestazione.
`-I`***percorso***	L'opzione '**-I**', a cui si attacca un percorso, aggiunge quel percorso a quelli usati per l'inclusione dei file di intestazione. Possono essere specificati più percorsi ripetendo l'uso dell'opzione.

L'esempio seguente mostra la compilazione del file '`mio.c`', per produrre il file '`mio.s`', contenente il codice in linguaggio assemblatore. Per la compilazione, i file di intestazione vengono cercati esclusivamente in percorsi stabiliti: '`../include`' e '`../../include`'.

```
$ bcc -ansi -0 -S -o mio.s ↩
```

```
↪     -I -I../include -I../../include mio.c[Invio]
```

L'esempio successivo è simile, ma si produce il file oggetto '`mio.o`':

```
$ bcc -ansi -0 -c -o mio.s ↩
↪     -I -I../include -I../../include mio.c[Invio]
```

I nomi delle variabili e delle funzioni scritte in linguaggio C, si traducono nel linguaggio assemblatore in nomi preceduti dal trattino basso. Per esempio, la funzione ***main()***, diventa il simbolo '**`_main`**'. Per questa ragione, quando si scrivono porzioni di codice in linguaggio assemblatore da esportare, occorre ricordare di aggiungere un trattino basso all'inizio.

A meno di voler produrre programmi per il sistema operativo ELKS, il compilatore Bcc va utilizzato con l'opzione '**`-c`**', oppure '**`-S`**', per poter controllare i passaggi successivi, in particolare la fase di collegamento dei vari componenti.

89.3.3 As86

As86 è un assemblatore in linguaggio x86, adatto alla compilazione di quanto prodotto da Bcc. La sintassi usata da As86 è fondamentalmente quella Intel.

`as86` [***opzioni***] ***file_s***

Tabella 89.53. Alcune opzioni per l'uso di As86.

Opzione	Descrizione
`-0`	Compila nella modalità a 16 bit e avvisa quando incontra istruzioni che non sono adatte a CPU 8086/8088.
`-o` ***nome***	Produce un file oggetto con il nome specificato.
`-s` ***nome***	Produce un file di testo contenente l'elenco dei simboli individuati. Questa opzione può essere usata assieme a '**`-o`**'.
`-u`	Fa in modo che i simboli privi di una dichiarazione siano importati dall'esterno senza specificare il segmento.

L'esempio seguente rappresenta una situazione di utilizzo comune, in cui si produce il file oggetto '`mio.o`', a partire dal sorgente '`mio.s`':

$ **as86 -u -0 -o mio.o mio.s** [*Invio*]

Come precisato a proposito di Bcc, se si devono importare dei simboli dal codice C, occorre aggiungere un trattino basso all'inizio dei nomi.

89.3.4 Ld86

Ld86 è il «collegatore» (*linker*) associato a As86. La caratteristica di Ld86 è quella di poter produrre un eseguibile «impuro» (come viene definito nella sua pagina di manuale), per il quale il segmento usato dal codice è lo stesso usato per i dati (***CS==DS==SS***), oppure può tenere separati il codice e i dati in segmenti distinti (ma in tal caso vale ancora l'uguaglianza ***DS==SS***).

Nella pagina di manuale di Ld86 si parla di «I&D», ovvero di istruzioni e dati, che possono essere separati o meno.

```
ld86 [opzioni] file_o...
```

Tabella 89.54. Alcune opzioni per l'uso di Ld86.

Opzione	Descrizione
`-d`	Elimina l'intestazione dal file che va a essere generato. L'utilizzo di questa opzione implica l'uso di '**-s**'.
`-s`	Elimina i simboli.
`-o` ***nome***	Produce un file eseguibile con il nome specificato.
`-i`	Tiene separati il segmento usato dal codice rispetto a quello dei dati. In mancanza di questa opzione, il segmento è lo stesso.

L'esempio seguente mostra la creazione di un programma, privo di intestazione e di simboli, dove tutto viene così definito attraverso il codice in modo esplicito. In particolare, si presume che il file '`crt0.o`' sia realizzato in modo da definire esattamente la forma della prima parte del file eseguibile.

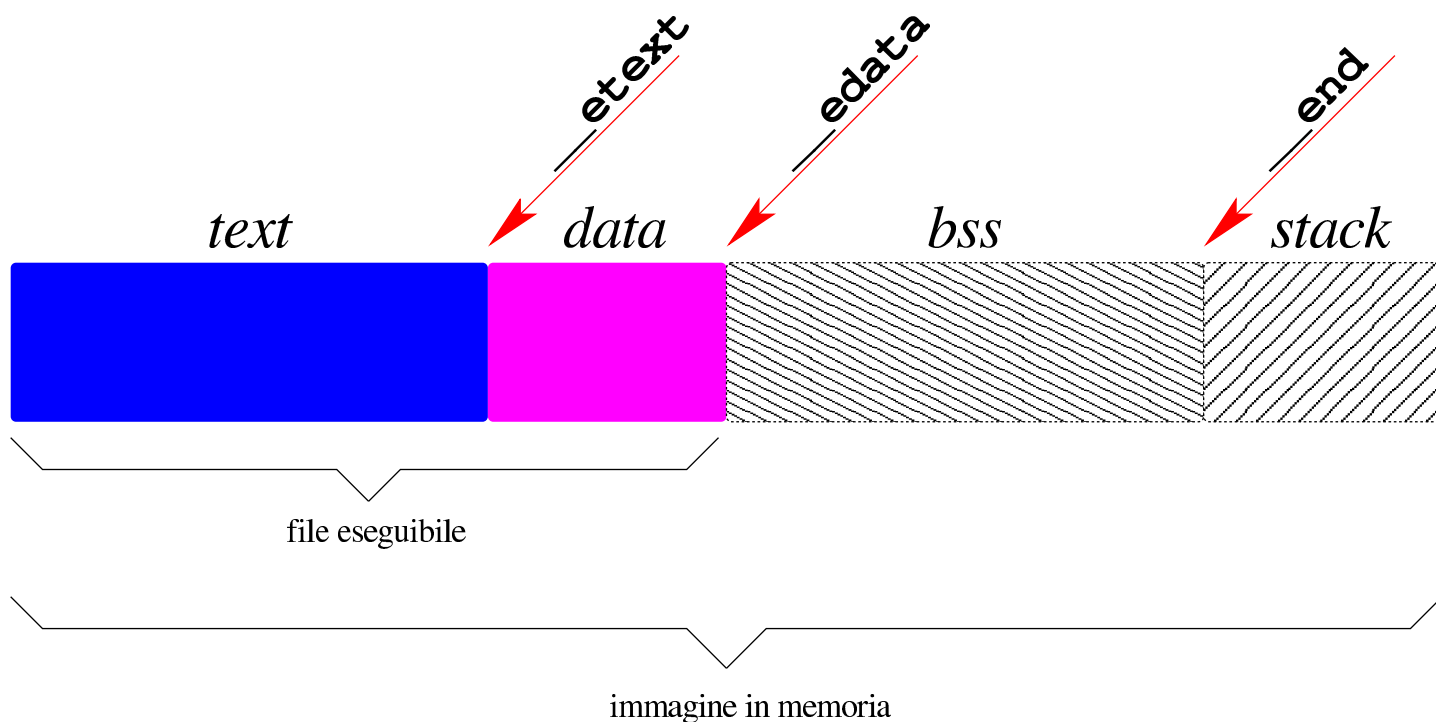
$ **ld86 -d -s -o kimage crt0.o body.o main.o** [*Invio*]

Ld86 definisce implicitamente dei simboli, raggiungibili dal codice che si scrive. Tra questi, sono molto importanti quelli seguenti, con cui è possibile determinare

la collocazione in memoria del programma, distinguendo tra codice e dati:

Simbolo per As86	Nome per Bcc	Descrizione
`__etext`	`_etext`	Variabile a 16 bit contenente l'indirizzo conclusivo dell'area usata dal codice, nell'ambito del segmento in cui si colloca.
`__edata`	`_edata`	Variabile a 16 bit contenente l'indirizzo conclusivo dell'area usata dai dati inizializzati, nell'ambito del segmento in cui si colloca.
`__end`	`_end`	Variabile a 16 bit contenente l'indirizzo conclusivo dell'area usata dai dati non inizializzati (BSS), esclusa la pila, nell'ambito del segmento in cui si colloca.
`__segoff`	`_segoff`	Variabile a 16 bit contenente la distanza tra l'inizio del segmento codice e l'inizio di quello usato per i dati, espressa in multipli di 16 bit. In presenza di eseguibili in cui il segmento è lo stesso, questo valore è zero.

In generale, la struttura di un file eseguibile prodotto da Ld86 è composta inizialmente dal codice, quindi continua con i dati inizializzati. Lo spazio dei dati non inizializzati non fa parte del file e deve essere previsto quando si carica il file in memoria, per metterlo in esecuzione; inoltre, lo stesso va fatto per la pila (*stack*) dei dati, la quale deve collocarsi dopo tale area.

Figura 89.56. Contenuto di un programma privo di intestazione e di simboli.

89.3.5 Bootblocks

Il pacchetto Bootblocks[3] consente di avviare un sistema per elaboratori x86-16, contenuto in un dischetto o in una partizione del disco fisso, con il kernel inserito nello stesso file system. Il pacchetto viene distribuito assieme agli strumenti di sviluppo Dev86, ma non viene compilato automaticamente assieme a quelli. Si trova precisamente nella sottodirectory '`bootblocks/`' dei sorgenti di Dev86. Si compila in modo molto semplice con il comando '**`make`**':

\# **`cd`** ***sorgenti_dev86*`/bootblocks`** [*Invio*]

\# **`make`** [*Invio*]

Dalla compilazione si ottengono diversi file e sono utili in particolare:

File	Descrizione
'**`makeboot`**'	programma per l'installazione del settore di avvio, da usare attraverso un sistema GNU/Linux comune;
'**`makeboot.com`**'	programma analogo a '**`makeboot`**', da usare con un sistema Dos.

Per avviare un programma autonomo o un kernel vero e proprio, in un dischetto con file system Minix 1, è sufficiente copiare tale file in modo che si trovi nella directory principale con il nome '`boot`', oppure si crea la directory '`/boot/`' e vi si colloca il file con il nome che si preferisce.

Supponendo di utilizzare un sistema GNU/Linux, supponendo di avere preparato il dischetto Minix (con i nomi al massimo di 14 byte) contenente tutto quello che serve, soprattutto con la directory '`/boot/`' o con il file '`boot`', se questo dischetto risulta inserito nell'unità corrispondente al file di dispositivo '`/dev/fd0`', **senza essere stato innestato**, si può eseguire il comando seguente, tenendo conto che il programma '**`makeboot`**' si presume collocato in una directory prevista tra i vari percorsi della variabile di ambiente '**`PATH`**':

`#` **`makeboot minix /dev/fd0`** [*Invio*]

```
Wrote sector 0
Wrote sector 1
```

Se il programma si accorge che il settore di avvio del dischetto contiene già qualcosa, si rifiuta di procedere, a meno di usare l'opzione '**`-f`**':

`#` **`makeboot -f minix /dev/fd0`** [*Invio*]

```
Boot block isn't empty, zap it first
Wrote sector 0
Wrote sector 1
```

È importante sapere che questo programma di avvio colloca in memoria il programma o il kernel da avviare a partire dall'indirizzo efficace 10000_{16}. Pertanto, dopo l'avvio effettivo, rimane inutilizzato lo spazio di memoria da 00500_{16} $0FFFF_{16}$.

89.4 Riferimenti

- Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, **prima edizione**, 1987, Prentice-Hall, ISBN 0-13-637406-9

  Appendice B: *introduction to the IBM PC*

- MAD, *Assembly tutorial*
  ⟨*http://www.xs4all.nl/~smit/asm01001.htm*⟩

- Wikipedia, *x86 instruction listings*
  ⟨*http://en.wikipedia.org/wiki/X86_instruction_listings*⟩

- *The x86 Interrupt List, aka "Ralf Brown's Interrupt List", "RBIL"*
  ⟨*http://www.cs.cmu.edu/~ralf/files.html*⟩

- *Computer interrupt*
  ⟨*http://wayback.archive.org/web/20040101000000*/http://calab.kaist.ac.kr/~hyoon/courses/cs310_2001fall/micro17.ppt*⟩

⟨*http://www.ece.msstate.edu/~reese/EE3724/lectures/interrupt/interrupt.pdf* ⟩

- BiosCentral, *BIOS data area*
  ⟨*http://www.bioscentral.com/misc/bda.htm* ⟩
- Robert de Bath, *Linux 8086 development environment*
  ⟨*http://homepage.ntlworld.com/robert.debath/* ⟩
  ⟨*http://homepage.ntlworld.com/robert.debath/dev86/* ⟩

[1] **Bcc, As86, Ld86** GNU GPL

[2] Va specificato che si tratta del compilatore Bcc di Bruce Evans, perché con questo nome o con questa sigla si trovano più facilmente riferimenti a compilatori C diversi, per esempio quello di Borland.

[3] **Bootblocks** GNU GPL

Studio per un sistema a 16 bit

90.1 Introduzione a os16

os16 è uno studio che applica qualche rudimento relativo ai sistemi operativi, basandosi sull'architettura x86-16 del vecchio IBM PC, utilizzando come strumenti di sviluppo Bcc, As86 e Ld86 (oltre a GNU GCC per controllare meglio la sintassi del codice C), su un sistema GNU/Linux. Il risultato non è un sistema operativo utilizzabile, ma una struttura su cui poter fare esperimenti e di cui è possibile mostrare (in termini tipografici) ed eventualmente descrivere ogni riga di codice.

os16 contiene uno schedulatore banale e molto limitato, un'organizzazione dei processi ad albero e una funzionalità limitata di amministrazione dei segnali, una gestione del file system Minix 1 (ma di unità intere, senza partizioni), una shell banale e qualche programma di servizio di esempio.

Per poter giungere rapidamente a un risultato e comunque per semplificare il codice, os16 utilizza le funzioni del BIOS tradizionale, le quali hanno lo svantaggio di impegnare in modo esclusivo l'elaboratore nel momento del loro funzionamento (dato che non possono essere rientranti). È noto che un sistema operativo multiprogrammato dignitoso non può avvalersi di tali funzionalità; pertanto, se si vuole studiare os16, non va dimenticato questo principio, benché qui sia stato trascurato.

Il kernel di os16 è monolitico, nel senso che incorpora tutte le proprie funzioni in un solo programma. Purtroppo, la dimensione del codice del kernel (e di qualunque altro processo di os16) non può superare i 64 Kibyte, ma la sua dimensione è già molto vicina a tale valore. Pertanto, non è possibile aggiungere funzionalità a questo sistema. Si può osservare che anche ELKS (⟨*http://elks.sourceforge.net/*⟩) soffra dello stesso limite e, d'altra parte, si può apprezzare come Minix 2 (⟨*http://minix1.woodhull.com/*⟩) riesca a superarlo attraverso un'organizzazione a microkernel.

90.1.1 Organizzazione

«

Tutti i file di os16 dovrebbero essere disponibili a partire da ⟨*allegati/os16*⟩. In particolare i file '`floppy.a`' e '`floppy.b`' sono le immagini di due dischetti da 1440 Kibyte, contenenti un file system Minix 1: il primo predisposto attraverso Bootblocks (sezione 89.3.5) per l'avvio di un kernel denominato '`boot`'; il secondo usato per essere innestato nella directory '`/usr/`' del primo.

Gli script preparati per lo sviluppo di os16 prevedono che i file-immagine dei dischetti vadano innestati nelle directory '`/mnt/os16.a/`' e '`/mnt/os16.b/`'. Pertanto, se si ricompila os16, tali directory vanno predisposte (oppure vanno modificati gli script con l'organizzazione che si preferisce attuare).

Per la verifica del funzionamento del sistema, è previsto l'uso equivalente di Bochs o di Qemu. Per questo scopo sono disponibili gli script '`bochs`' e '`qemu`' (rispettivamente i listati 100.1.1 e 100.1.2), con le opzioni necessarie a operare correttamente.

Per la compilazione del lavoro si usa lo script '`makeit`' (listato 100.1.3), il quale ricrea ogni volta i file-make, basandosi sui file presenti effettivamente nelle varie directory previste. Questo script, alla fine della compilazione, copia il kernel nel file-immagine del primo dischetto (purché risulti innestato come previsto nella directory '`/mnt/os16.a/`') e con esso copia anche gli applicativi principali, mentre il resto viene copiato nel secondo.

Nello script 'makeit', la variabile di ambiente '**TAB**' deve contenere esattamente il carattere di tabulazione (<*HT*>), corrispondente al codice 09_{16}. Il file che viene distribuito contiene invece l'assegnamento di uno spazio puro e semplice, che va modificato a mano, sostituendolo con tale codice. La riga da modificare è quella in cui la variabile viene dichiarata:

```
local TAB=" "
```

Va osservato che il lavoro si basa su un file system Minix 1 (sezione 74.7) perché è molto semplice, ma soprattutto, la prima versione è quella che può essere utilizzata facilmente in un sistema operativo GNU/Linux (sul quale avviene lo sviluppo di os16). È bene sottolineare che si tratta della versione con nomi da 14 caratteri, ovvero quella tradizionale del sistema operativo Minix, mentre nei sistemi GNU/Linux, la creazione predefinita di un file system del genere produce una versione particolare, con nomi da 30 caratteri.

90.1.2 Le directory

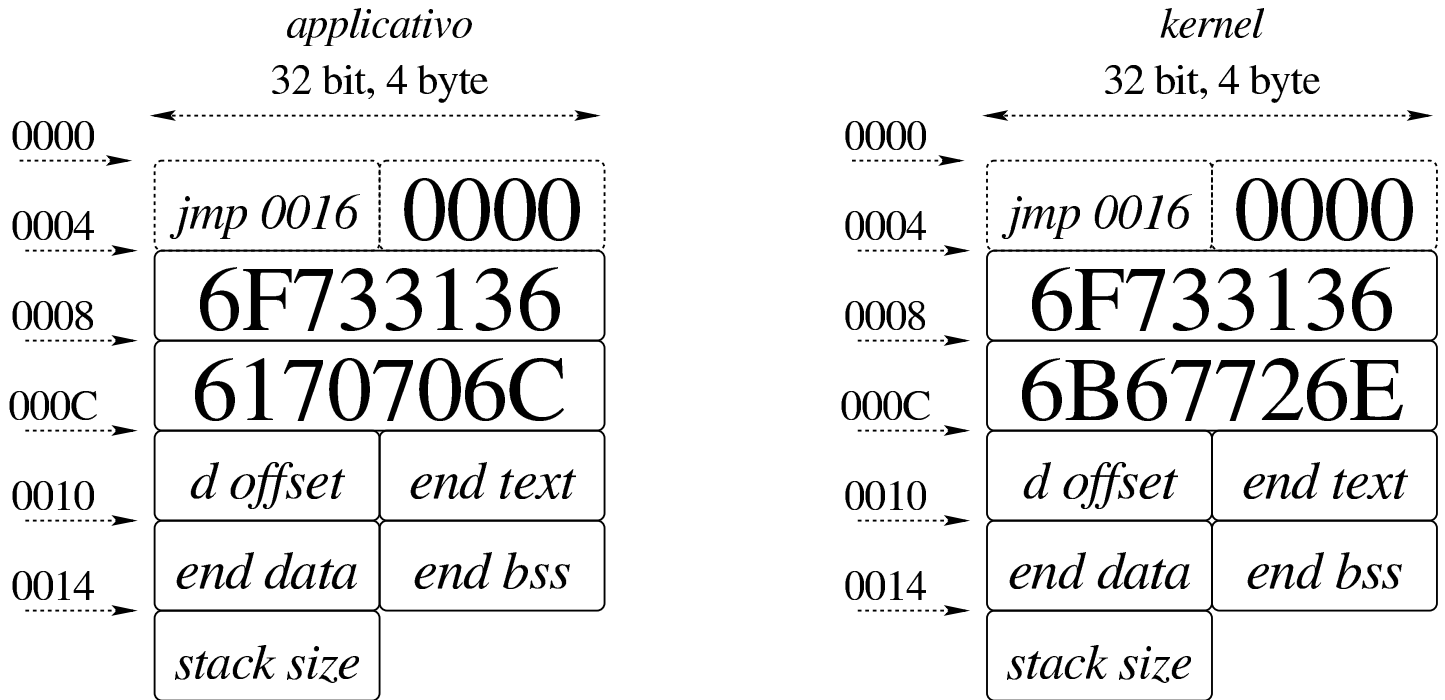
Gli script descritti nella sezione precedente, si trovano all'inizio della gerarchia prevista per os16. Le directory successive dividono in modo molto semplice le varie componenti per la compilazione:

Directory	Contenuto
'applic/'	File delle applicazioni da usare con os16.
'kernel/'	File per la realizzazione del kernel, inclusi i file di intestazione specifici.
'lib/'	File di intestazione generali, file della libreria C per le applicazioni e, per quanto possibile, anche per il kernel.
'ported/'	Applicativi di altri autori, adattati per os16.
'skel/'	Scheletro del file system complessivo, con i file di configurazione e le pagine di manuale.

La libreria C non è completa, limitandosi a contenere ciò che serve per lo stato di avanzamento attuale del lavoro. Si osservi che nella directory 'lib/bcc/' si collocano file contenenti una libreria di funzioni in linguaggio assemblatore, necessaria al compilatore Bcc per compiere il proprio lavoro correttamente con valori da 32 bit. Tale libreria è scritta dall'autore originale di Bcc, Bruce Evans, e viene inclusa in modo da garantire che la compilazione non richieda alcun file estraneo.

90.1.3 La struttura degli eseguibili

Nell'ottica della massima semplicità, gli eseguibili di os16 hanno un'intestazione propria, schematizzata dalla figura successiva. Tale intestazione viene ottenuta attraverso il file '`crt0.s`', che è comunque differente se si tratta di un applicativo o del kernel.

Figura 90.2. Struttura iniziale dei file eseguibili di os16.

Nella figura si mettono a confronto la parte iniziale dell'eseguibile di un applicativo con quella del kernel di os16. Nei primi quattro byte c'è un'istruzione di salto al codice che si trova subito dopo l'intestazione, quindi appare un'impronta di riconoscimento che occupa quattro byte. Tale impronta è la rappresentazione esadecimale della stringa «os16». Successivamente appare un'altra impronta, con cui si distingue se si tratta di un applicativo o del kernel; si tratta in pratica della sequenza di «appl», oppure di «kern». Tuttavia, a causa dell'inversione dell'ordine dei byte, in pratica, se si visualizza il file binario si legge «61so», «lppa» e «nrek».

Dopo l'impronta di riconoscimento si trovano, rispettivamente, lo scostamento del segmento dati, espresso in multipli di 16 (in pratica, 1234_{16} rappresenterebbe uno scostamento di 12340_{16} byte, rispetto all'inizio del codice), gli indirizzi conclusivi dell'area del codice, dei dati inizializzati e di quelli non inizializzati. Alla fine viene indicata la dimensione richiesta per la pila dei dati. Ciò che appare dopo è il codice del programma.

Il kernel e gli applicativi di os16 sono compilati in modo da rendere indipendenti l'area del codice rispetto a quella dei dati (si dice che hanno aree «I&D separate»).

Nel caso del kernel, le informazioni successive alle impronte di riconoscimento non vengono utilizzate, perché il kernel viene collocato in uno spazio preciso in memoria:

l'area dati va a trovarsi dall'indirizzo efficace 00500_{16} fino a $104FF_{16}$ incluso, mentre l'area del codice inizia da 10500_{16} fino alla fine.

Per fare in modo che il proprio sistema GNU possa riconoscere correttamente questi file, si può modificare la configurazione del file '`/etc/magic`', aggiungendo le righe seguenti:

```
4       quad       0x6B65726E6F733136      os16 kernel
4       quad       0x6170706C6F733136      os16 application
```

90.1.4 Caricamento del kernel

Il kernel è preparato per trovarsi inizialmente in memoria, tale e quale al file da cui viene caricato, a partire dall'indirizzo efficace 10000_{16}, così come avviene quando si utilizza Bootblocks (a cui si è già accennato nel capitolo). Successivamente il kernel stesso si sposta, copiandosi inizialmente a partire dall'indirizzo efficace 30000_{16}, quindi suddividendosi e mettendo la propria area dati a partire dall'indirizzo 00500_{16} e l'area codice da 10500_{16} (lo spazio di memoria che va da 00000_{16} a $004FF_{16}$ incluso, non può essere utilizzato, perché contiene la tabella IVT e l'area BDA, secondo l'architettura degli elaboratori IBM PC tradizionali).

90.1.5 Informazioni diagnostiche

Nel codice del kernel vengono usate, in varie occasioni, delle funzioni che hanno lo scopo di visualizzare delle informazioni diagnostiche. Queste funzioni sono raccolte in file della directory '`kernel/diag/`', a cui si abbina il file di intestazione '`kernel/diag.h`' (listato 100.3 e successivi). In generale, in questa documentazione, non viene dato molto spazio alla descrizione di queste funzioni, perché hanno un ruolo marginale e sono fatte per essere modificate in base alle esigenze di verifica del momento.

90.1.6 Tabelle

Nel codice del kernel si utilizzano spesso delle informazioni organizzate in memoria in forma di tabella. Si tratta precisamente di array, le cui celle sono costituite generalmente da variabili strutturate. Queste tabelle, ovvero gli array che le rappresentano, sono dichiarate come variabili pubbliche; tuttavia, per facilitare l'accesso ai rispettivi elementi e per uniformità di comportamento, viene abbinata loro una funzione, con un nome terminante per '...**`_reference()`**', con cui si ottiene il puntatore a un certo elemento della tabella, fornendo gli argomenti appropriati. Per esempio, la tabella degli inode in corso di utilizzazione viene dichiarata così nel file '`kernel/inode/inode_table.c`':

```
inode_t inode_table[INODE_MAX_SLOTS];
```

Successivamente, la funzione ***inode_reference()*** offre il puntatore a un certo inode:

```
inode_t *inode_reference (dev_t device, ino_t ino);
```

90.1.7 Guida di stile

«

Per cercare di dare un po' di uniformità al codice del kernel e a quello della libreria, dove possibile, i nomi delle variabili seguono una certa logica, riassunta dalla tabella successiva.

Tipo	Nome	Utilizzo
`inode_t *`	`inode` `inode_...`	Puntatore a un inode (puntatore a un elemento della tabella di inode).
`ino_t`	`ino` `ino_...`	Numero di inode, nell'ambito di un certo super blocco (ammesso che sia abbinato effettivamente a un dispositivo).
`int`	`fdn` `fdn_...`	Numero del descrittore di un file (indice all'interno della tabella dei descrittori).
`fd_t *`	`fd` `fd_...`	Puntatore a un descrittore di file (puntatore a un elemento della tabella di descrittori).
`int`	`fno` `fno_...`	Numero del file di sistema (indice all'interno della tabella dei file di sistema).
`zno_t`	`zone` `zone_...`	Numero assoluto di una «zona» del file system Minix.
`zno_t`	`fzone` `fzone_...`	Numero relativo di una «zona» del file system Minix. In questo caso, il numero della zona è relativo al file, dove la prima zona del file ha il numero zero.

Tipo	Nome	Utilizzo

`off_t`	`offset`
	`offset_...`
	`off_...`
		Scostamento, secondo il significato del tipo derivato '**`off_t`**'.

`size_t` `ssize_t`	`size` `size_...`	Dimensione, secondo il significato dei tipi derivati '**`size_t`**' o '**`ssize_t`**'.
`size_t` `ssize_t`	`count` `count_...`	Quantità, quando il tipo '**`size_t`**' è appropriato.
`blkcnt_t`	`blkcnt` `blkcnt_...`	Quantità espressa in blocchi del file system (in questo caso, trattandosi di un file system Minix 1, si intendono zone).
`blksize_t`	`blksize` `blksize_...`	Dimensione del blocco del file system, espressa in byte (in questo caso, trattandosi di un file system Minix 1, si intende la dimensione della zona).
`int`	`fno` `fno_...`	Numero di file system.
`int`	`oflags` `oflags_...`	Opzioni relative all'apertura di un file, annotate nella tabella dei file di sistema: indicatori di sistema.
`int`	`status` `status_...`	Valore intero restituito da una funzione, quando la risposta contiene solo l'indicazione di un successo o di un insuccesso.
`void *`	`pstatus`	Puntatore restituito da una funzione, quando interessa sapere solo se si tratta di un esito valido.

Tipo	Nome	Utilizzo
char *	path path_...	Percorso del file system.
dev_t	device device_...	Numero di dispositivo, contenente sia il numero primario, sia quello secondario (*major*, *minor*).
int	n n_...	Dimensione di qualcosa, di tipo '**int**'.
char *	string string_...	Area di memoria da considerare come stringa.
void *	buffer buffer_...	Area di memoria destinata ad accogliere un'informazione di tipo imprecisato.
int	n n_...	Dimensione o quantità di qualcosa, espressa attraverso il tipo '**int**'.
int	c c_...	Un carattere senza segno trasformato nel tipo '**int**'.
struct stat	st st_...	Variabile strutturata usata per rappresentare lo stato di un file, secondo il tipo '**struct stat**'.
FILE *	fp fp_...	Puntatore che rappresenta un flusso di file.

Tipo	Nome	Utilizzo
`DIR *`	`dp` `dp_...`	Puntatore che rappresenta un flusso relativo a una directory.
`struct dirent`	`dir` `dir_...`	Variabile strutturata contenente le informazioni su una voce di una directory.
`struct password`	`pws` `pws_...`	Variabile strutturata contenente le informazioni di una voce del file '`/etc/passwd`'.
`struct tm`	`tms` `tms_...`	Variabile strutturata contenente le componenti di un orario.
`struct tm *`	`timeptr` `timeptr_...`	Puntatore a una variabile strutturata contenente le componenti di un orario.

90.1.8 Tipi derivati speciali

Nel codice del kernel si usano dei tipi derivati speciali, riassunti nella tabella successiva.

File di intestazione	Tipo speciale	Descrizione
'`kernel/memory.h`'	`addr_t`	Variabile scalare, in grado di rappresentare un indirizzo efficace di memoria (un indirizzo che vada da 00000_{16} a $FFFFF_{16}$).
'`kernel/memory.h`'	`segment_t`	Variabile scalare, a 16 bit, usata per rappresentare il valore di un registro di segmento.

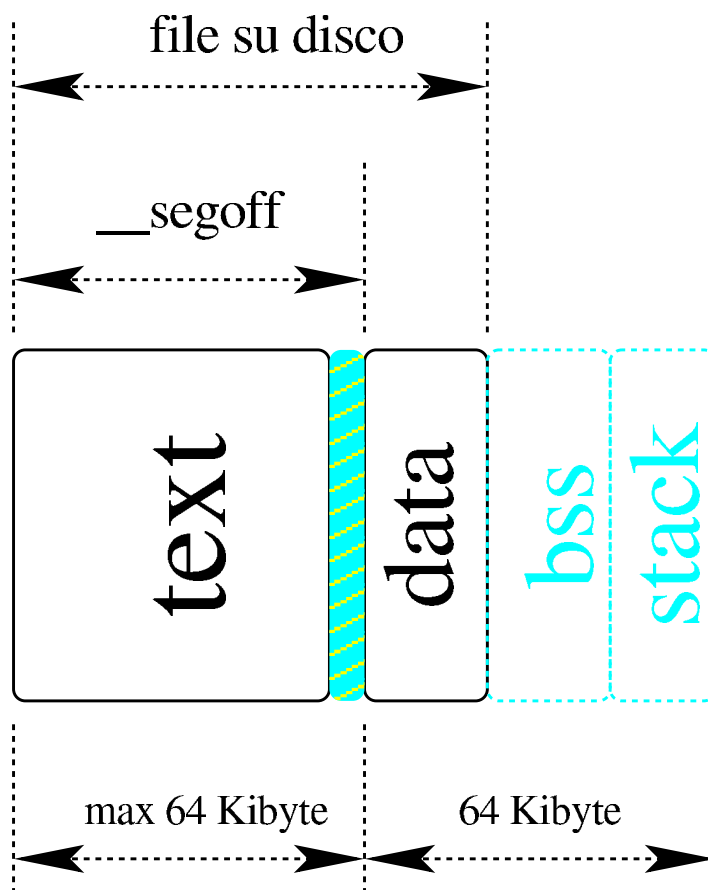
File di intestazione	Tipo speciale	Descrizione
'kernel/memory.h'	offset_t	Variabile scalare, a 16 bit, usata per rappresentare lo scostamento di memoria, a partire dall'inizio di un segmento. Questo tipo di variabile non va confuso con il tipo '**off_t**', il quale è un valore con segno e di rango maggiore rispetto a questo '**offset_t**'.
'kernel/memory.h'	memory_t	Variabile strutturata, adatta a contenere tutte le coordinate utili a individuare una certa area di memoria, secondo l'architettura prevista da os16.
'kernel/tty.h'	tty_t	Variabile strutturata, adatta a contenere le informazioni e lo stato di un terminale.
'kernel/ibm_i86.h'	dsk_t	Variabile strutturata, adatta a contenere le informazioni hardware di una certa unità di memorizzazione.
'kernel/ibm_i86.h'	dsk_chs_t	Variabile strutturata, adatta a contenere le coordinate di un certo settore (cilindro, testina e settore) in un'unità di memorizzazione.
'kernel/fs.h'	zno_t	Variabile scalare, per rappresentare un numero di una zona, secondo la terminologia del file system Minix.
'kernel/fs.h'	sb_t	Variabile strutturata, adatta a contenere tutte le informazioni di un super blocco, relativo a un dispositivo di memorizzazione innestato.
'kernel/fs.h'	inode_t	Variabile strutturata, adatta a contenere tutte le informazioni di un inode aperto nel sistema.
'kernel/fs.h'	file_t	Variabile strutturata, adatta a contenere i dati di un file di sistema.

File di intestazione	Tipo speciale	Descrizione
‘kernel/fs.h’	fd_t	Variabile strutturata, adatta a contenere i dati di un descrittore di file, ovvero del file di un certo processo elaborativo.
‘kernel/fs.h’	directory_t	Variabile strutturata, adatta a contenere una voce di una directory.

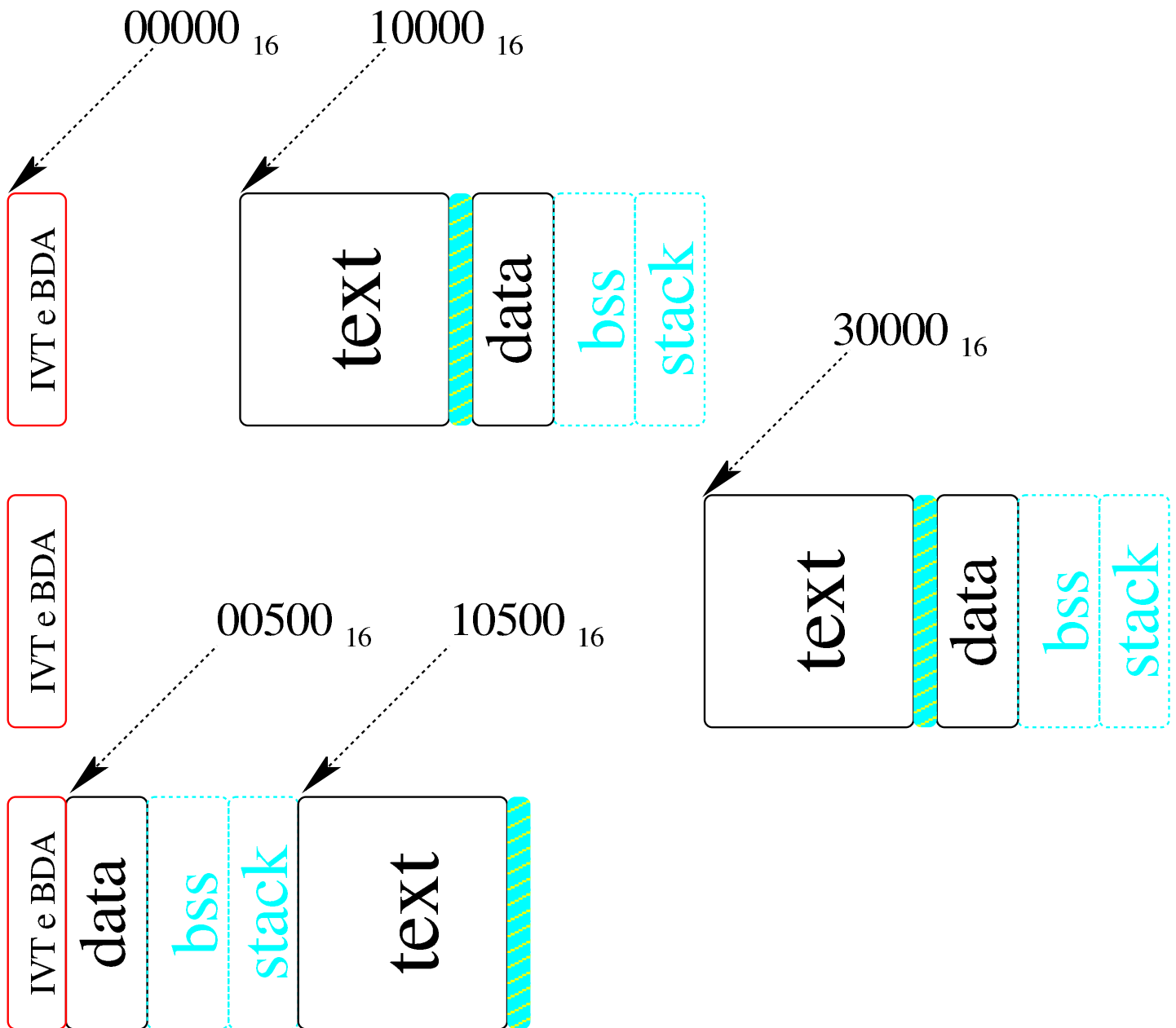
90.2 Caricamento ed esecuzione del kernel

Il kernel di os16 (ma così vale anche per gli applicativi) viene compilato senza un’intestazione predefinita, pertanto questa viene costruita nel primo file: ‘crt0.s’. Questo file ha lo scopo di eseguire la funzione ***main()*** del kernel, in cui si sintetizza il funzionamento dello stesso.

90.2.1 Dal file su disco alla copia in memoria

Il file del kernel prodotto dagli strumenti di sviluppo è strutturato come sintetizza il disegno seguente:

La prima parte del file è utilizzata dal codice (*text*), quindi ci può essere un piccolissimo spazio inutilizzato, seguito dalla porzione che riguarda i dati, tenendo conto che

nel file ci sono solo i dati inizializzati, mentre gli altri non hanno bisogno di essere rappresentati, ma in memoria occupano comunque il loro spazio.

Il kernel è organizzato per tenere separate l'area delle istruzioni da quella dei dati, pertanto il compilatore (precisamente il «collegatore», ovvero il *linker*) offre il simbolo ***__segoff***, con il quale si conosce la distanza del segmento dei dati dall'inizio del file. Il valore di questo scostamento è espresso in «paragrafi», ovvero in multipli di 16; in pratica si tratta dello scostamento da utilizzare in un registro di segmento. Dal momento che lo scostamento effettivo è costituito dalla dimensione dell'area del codice, approssimata per eccesso ai 16 byte successivi, tra la fine dell'area codice e l'inizio di quella dei dati c'è quel piccolo spazio vuoto a cui già si è fatto riferimento.

Il kernel viene caricato in memoria, con l'ausilio di Bootblocks, all'indirizzo 10000_{16}. Da lì il kernel si mette in funzione e, prima si copia all'indirizzo 30000_{16}, quindi riprende a funzionare dal nuovo indirizzo, poi si copia mettendo i dati a partire dall'indirizzo 00500_{16} (dopo la tabella IVT e dopo l'area BDA) e il codice a partire

dall'indirizzo 10500_{16}. Alla fine, riprende a funzionare dall'indirizzo 10500_{16}. La pila dei dati (*stack*) viene attivata solo quando il kernel ha trovato la sua collocazione definitiva.

90.2.2 File «kernel/main/crt0.s»

Listato 100.7.2.

Dopo il preambolo in cui si dichiarano i simboli esterni e quelli interni da rendere pubblici, con l'istruzione '**`entry startup`**' si dichiara all'assemblatore che il punto di partenza è costituito dal simbolo '**`startup`**', ma in ogni caso questo deve essere all'inizio del codice, mancando un'intestazione precostituita. In pratica, la primissima cosa che si ottiene nel file eseguibile finale è un'istruzione di salto a una posizione più avanzata del codice, dove si colloca il simbolo '**`startup_code`**', e nello spazio intermedio (tra quell'istruzione di salto e il codice che si trova a partire da '**`startup_code`**') si collocano le impronte di riconoscimento, oltre ai dati sulla dislocazione dell'eseguibile in memoria.

```
...
entry startup
...
startup:
    jmp startup_code
...
startup_code:
...
```

Tra la prima istruzione di salto e le impronte di riconoscimento, introdotte dal simbolo '**`magic`**', c'è uno spazio vuoto (nullo), calcolato automaticamente in modo da garantire che la prima impronta inizi all'indirizzo relativo 0004_{16}. Di seguito vengono gli altri dati.

```
...
startup:
    jmp startup_code
filler:
    .space (0x0004 - (filler - startup))
magic:
    .data4 0x6F733136
    .data4 0x6B65726E
segoff:
    .data2 __segoff
```

```
etext:
    .data2 __etext
edata:
    .data2 __edata
ebss:
    .data2 __end
stack_size:
    .data2 0x0000
.align 2
startup_code:
...
```

A partire da '**`startup_code`**' viene analizzato il valore effettivo del registro ***CS***. Se questo è pari a 1000_{16}, significa che il kernel si trova in memoria a partire dall'indirizzo efficace 10000_{16}, ma in tal caso si salta a una procedura che copia il kernel in un'altra posizione di memoria (30000_{16}); se invece il valore di ***CS*** viene riconosciuto pari a quello della destinazione della prima copia, si passa a un'altra procedura che scompone l'area dati e l'area codice (testo) del kernel, in modo da collocare l'area dati a partire da 00500_{16} e l'area codice a partire da 10500_{16}. Quando si riconosce che il valore di ***CS*** è quello finale, si salta al simbolo '**`main_code`**' e da lì inizia il lavoro vero e proprio.

```
...
startup_code:
    mov cx, cs
    xor cx, #0x1000
    jcxz move_code_from_0x1000_to_0x3000
    mov cx, cs
    xor cx, #0x3000
    jcxz move_code_from_0x3000_to_0x0050
    mov cx, cs
    xor cx, #0x1050
    jcxz main_code
    hlt
    jmp startup_code
move_code_from_0x1000_to_0x3000:
    ...
    jmp far #0x3000:#0x0000
move_code_from_0x3000_to_0x0050:
```

```
    …
    jmp far #0x1050:#0x0000
main_code:
…
```

Non si prevede che il kernel possa trovarsi in memoria in una collocazione differente da quelle stabilite nelle varie fasi di avvio, pertanto, in caso contrario, si crea semplicemente un circolo vizioso senza uscita.

Dal simbolo '**`main_code`**' inizia finalmente il lavoro e si procede con l'allineamento dei registri dei segmenti dei dati, in modo che siano tutti corrispondenti al valore previsto: 0050_{16} (il segmento in cui inizia l'area dati, secondo la collocazione prevista). Viene poi posizionato il valore del registro ***SP*** a zero, in modo che al primo inserimento questo punti esattamente all'indirizzo più grande che si possa raggiungere nel segmento dati ($FFFE_{16}$, considerato che gli inserimenti nella pila sono a 16 bit).

```
…
main_code:
    mov  ax, #0x0050
    mov  ds, ax
    mov  ss, ax
    mov  es, ax
    mov sp, #0x0000
    …
```

Appena la pila diventa operativa, si inizializza anche il registro ***FLAGS***, verificando di disabilitare inizialmente le interruzioni.

```
…
main_code:
    …
    push #0
    popf
    cli
    …
```

A questo punto, si chiama la funzione ***main()***, fornendo come argomenti tre valori a zero.

```
...
main_code:
    ...
    push #0
    push #0
    push #0
    call _main
    add  sp, #2
    add  sp, #2
    add  sp, #2
...
```

Nel caso la funzione dovesse terminare e restituire il controllo, si passerebbe al codice successivo al simbolo '**`halt`**', con cui si crea un ciclo senza uscita, corrispondente alla conclusione del funzionamento del kernel.

```
...
halt:
    hlt
    jmp halt
...
```

Utilizzando il compilatore Bcc per compilare ciò che descrive la funzione ***main()***, viene richiesta la presenza della funzione ***__mkargv()*** (il simbolo '**`___mkargv`**'), che in questo caso può limitarsi a non fare alcunché.

```
...
___mkargv:
    ret
...
```

90.2.3 File «kernel/main.h» e «kernel/main/*»

«

Listato 100.7 e successivi.

Tutto il lavoro del kernel di os16 si sintetizza nella funzione ***main()***, contenuta nel file '`kernel/main/main.c`'. Per poter dare un significato a ciò che vi appare al suo interno, occorre conoscere tutto il resto del codice, ma inizialmente è utile avere un'idea di ciò che succede, se poi si vuole compilare ed eseguire il sistema operativo.

La funzione ***main()*** viene dichiarata secondo la forma tradizionale di un programma per sistemi POSIX, ma gli argomenti che riceve dalla chiamata contenuta nel file

'kernel/main/crt0.s' sono nulli, perché nessuna informazione gli viene passata effettivamente.

```
…
int
main (int argc, char *argv[], char *envp[])
{
    …
    tty_init ();
    k_printf ("os16 build %s ram %i Kibyte\n", BUILD_DATE,
              int12 ());
    dsk_setup ();
    heap_clear ();
    proc_init ();
    menu ();
    …
```

Dopo la dichiarazione delle variabili si inizializza la gestione del video della console con la funzione ***tty_init()***, si mostra un messaggio iniziale, quindi si passa alla predisposizione di ciò che serve, prima di poter avviare dei processi. In particolare va osservata la funzione ***heap_clear()***, la quale inizializza con il codice $FFFF_{16}$ lo spazio di memoria libero, tra la fine delle variabili «statiche» e il livello che ha raggiunto in quel momento la pila dei dati. Successivamente, avendo marcato in questo modo quello spazio, diventa possibile riconoscere empiricamente quanto spazio di quella porzione di memoria avrebbe potuto essere utilizzato, senza essere sovrascritto dalla pila dei dati. Il messaggio iniziale contiene la data di compilazione e la memoria libera (la macro-variabile ***BUILD_DATE*** viene definita dallo script '**makeit**', usato per la compilazione, creando il file 'kernel/main/build.h' che viene poi incluso dal file 'kernel/main/main.c').

L'attivazione della gestione dei processi (e delle interruzioni) con la funzione ***proc_init()***, comporta anche l'innesto del file system principale (chiamando da lì la funzione ***sb_mount()***).

```
…
int
main (int argc, char *argv[], char *envp[])
{
    …
    menu ();
    for (exit = 0; exit == 0;)
```

```
    {
      sys (SYS_0, NULL, 0);
      dev_io ((pid_t) 0, DEV_TTY, DEV_READ, 0L, &key, 1,
              NULL);
      ...
      switch (key)
        {
          case 'h':
              menu ();
              break;
          ...
          case 'x':
              exit = 1;
              break;
          case 'q':
              k_printf ("System halted!\n");
              return (0);
              break;
        }
    }
...
```

A questo punto il kernel ha concluso le sue attività preliminari e, per motivi diagnostici, mostra un menù, quindi inizia un ciclo in cui ogni volta esegue una chiamata di sistema nulla e poi legge un carattere dalla tastiera: se risulta premuto un tasto previsto, fa quanto richiesto e riprende il ciclo. La chiamata di sistema nulla serve a far sì che lo schedulatore ceda il controllo a un altro processo, ammesso che questo esista, consentendo l'avvio di processi ancor prima di avere messo in funzione quel processo che deve svolgere il ruolo di '**`init`**'.

In generale le chiamate di sistema sono fatte per essere usate solo dalle applicazioni; tuttavia, in pochi casi speciali il kernel le deve utilizzare come se fosse proprio un'applicazione. Qui si rende necessario l'uso della chiamata nulla, perché quando è in funzione il codice del kernel non ci possono essere interruzioni esterne e quindi nessun altro processo verrebbe messo in condizione di funzionare.

Le funzioni principali disponibili in questa modalità diagnostica sono riassunte nella tabella successiva:

Tasto	Risultato
[*h*]	Mostra il menù di funzioni disponibili.
[*1*]	Invia il segnale '**`SIGKILL`**' al processo numero uno.
[*2*]...[*9*]	Invia il segnale '**`SIGTERM`**' al processo con il numero corrispondente.
[*A*]...[*F*]	Invia il segnale '**`SIGTERM`**' al processo con il numero da 10 a 15.
[*a*], [*b*], [*c*]	Avvia il programma '`/bin/aaa`', '`/bin/bbb`' o '`/bin/ccc`'.
[*f*]	Mostra l'elenco dei file aperti nel sistema.
[*m*], [*M*]	Innesta o stacca il secondo dischetto dalla directory '`/usr/`'.
[*n*], [*N*]	Mostra l'elenco degli inode aperti: l'elenco è composto da due parti.
[*l*]	Invia il segnale '**`SIGCHLD`**' al processo numero uno.
[*p*]	Mostra la situazione dei processi e altre informazioni.
[*x*]	Termina il ciclo e successivamente si passa all'avvio di '`/bin/init`'.
[*q*]	Ferma il sistema.

Premendo [*x*], il ciclo termina e il kernel avvia '`/bin/init`'. Quindi si mette in un altro ciclo, dove si limita a passare ogni volta il controllo allo schedulatore, attraverso la chiamata di sistema nulla.

```
…
int
main (int argc, char *argv[], char *envp[])
{
    …
    menu ();
    for (exit = 0; exit == 0;)
      {
        …
      }
    exec_argv[0] = "/bin/init";
    exec_argv[1] = NULL;
    pid = run ("/bin/init", exec_argv, NULL);
    while (1)
      {
        sys (SYS_0, NULL, 0);
      }
    …
}
```

Figura 90.22. Aspetto di os16 in funzione, con il menù in evidenza.

```
os16 build 20YY.MM.DD HH:MM:SS ram 639 Kibyte
.--------------------------------------------------------------.
| [h]       show this menu                                     |
| [p]       process status and memory map                      |
| [1]..[9]  kill process  1 to 9                               |
| [A]..[F]  kill process 10 to 15                              |
| [l]       send SIGCHLD to process 1                          |
| [a]..[c]  run programs '/bin/aaa' to '/bin/ccc' in parallel  |
| [f]       system file status                                 |
| [n], [N]  list of active inodes                              |
| [m], [M]  mount/umount '/dev/dsk1' at '/usr/'                |
| [x]       exit interaction with kernel and start '/bin/init' |
| [q]       quit kernel                                        |
`--------------------------------------------------------------'
```

Figura 90.23. Aspetto di os16 in funzione mentre visualizza anche la tabella dei processi avviati (tasto [*p*]). OGV ⟨*http://www.youtube.com/watch?v=0gbgNpRrXBU*⟩

```
ababaaababaaababaabbaaababaaababap
pp  p pg
id id rp  tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name
 0  0  0 0000    0    0    0 00.35 R 10500 eb7c 00500 0000 ffc8 os16 kernel
 0  1  0 0000    0    0    0 00.33 r 2f100 0600 2f700 aa00 a8e8 /bin/ccc
 0  2  0 0000    0    0    0 00.01 r 1f100 0600 84300 aa00 a8e8 /bin/ccc
 2  3  0 0000   10   10   10 00.01 r 1f100 0600 44b00 aa00 a8e8 /bin/ccc
 0  4  0 0000    0    0    0 00.17 r 21600 0600 3a100 aa00 a8e8 /bin/ccc
 4  5  0 0000   10   10   10 00.02 r 21c00 2400 6f100 a900 a86c /bin/aaa
 4  6  0 0000   11   11   11 00.02 s 24000 2500 59d00 a900 a8b6 /bin/bbb
 0  7  0 0000    0    0    0 00.13 r 26500 0600 64600 aa00 a8e8 /bin/ccc
 7  8  0 0000   10   10   10 00.02 r 26b00 2400 8ee00 a900 981e /bin/aaa
 7  9  0 0000   11   11   11 00.02 s 2bf00 2500 79a00 a900 a8b6 /bin/bbb
CS=1050 DS=0050 SS=0050 ES=0050 BP=ffe4 SP=ffe4 heap_min=878c etext=eb7c edata=1
b3c ebss=4c34 ksp=ffc8 clock=0000084b, time elapsed=00:01:57
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffe00000003ffffffffffffffffffffffffff
fffe000000000001fffffffff0007fffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff800
0000000000000000000000000000000000000007ffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffff7fffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffbfffffffffffffffffffffffffffffffffffffffffe0000000000000000000000000f

abbaaaaababaaababaaabbaabbaabababbaabbbbbbbbbbbbbb
```

Figura 90.24. Aspetto di os16 in funzione con il menù in evidenza, dopo aver premuto il tasto [*x*] per avviare '**init**'. OGV ⟨*http://www.youtube.com/watch?v=epql4EhgWPU* ⟩

```
os16 build 20YY.MM.DD HH:MM:SS ram 639 Kibyte
.--------------------------------------------------------------------.
| [h]       show this menu                                           |
| [p]       process status and memory map                            |
| [1]..[9] kill process  1 to 9                                      |
| [A]..[F] kill process 10 to 15                                     |
| [l]       send SIGCHLD to process 1                                |
| [a]..[c] run programs '/bin/aaa' to '/bin/ccc' in parallel         |
| [f]       system file status                                       |
| [n], [N] list of active inodes                                     |
| [m], [M] mount/umount '/dev/dsk1' at '/usr/'                       |
| [x]       exit interaction with kernel and start '/bin/init'       |
| [q]       quit kernel                                              |
`--------------------------------------------------------------------'
init
os16: a basic os. [Ctrl q], [Ctrl r], [Ctrl s], [Ctrl t] to change
console.
This is terminal /dev/console0
Log in as "root" or "user" with password "ciao" :-)
login:
```

90.3 Funzioni interne legate all'hardware

Il file '`kernel/ibm_i86.h`' e quelli contenuti nella directory '`kernel/ibm_i86/`', raccolgono il codice del kernel che è legato strettamente all'hardware; a questi file vanno anche aggiunti '`lib/sys/os16.h`' e la directory '`lib/sys/os16/`', della libreria, utilizzati anche dalle applicazioni, a vario titolo. In generale si può osservare la presenza di funzioni che si avvalgono direttamente di alcune interruzioni del BIOS (fondamentalmente per la gestione del video e per l'accesso ai dischi); funzioni che permettono di leggere il valore di alcuni registri; funzioni per leggere e scrivere la memoria, in posizioni arbitrarie; funzioni per facilitare la lettura e la scrittura nei dischi, anche a livello di byte.

Salvo poche eccezioni, le funzioni scritte in linguaggio assemblatore hanno nomi che iniziano con un trattino basso, ma a fianco di queste sono anche disponibili delle macro-istruzioni, con nomi equivalenti, senza il trattino basso iniziale, per garantire che gli argomenti della chiamata abbiano il tipo corretto, restituendo un valore intero «normale», quando qualcosa deve essere restituito.

90.3.1 Libreria: «lib/sys/os16.h» e «lib/sys/os16/...»

«

Listato 101.12 e successivi.

Nel file ‘`lib/sys/os16.h`’ e in quelli della directory ‘`lib/sys/os16/`’ si raccolgono, tra le altre, delle funzioni di basso livello che possono essere utili per il kernel e per le applicazioni. Si tratta di ***_seg_i()*** e ***_seg_d()*** (ovvero le macro-istruzioni ***seg_i()*** e ***seg_d()***), con cui si ottiene, rispettivamente, il numero del segmento codice (istruzioni) e il numero del segmento dati. Inoltre, per poter verificare gli altri registri di segmento e i registri di gestione della pila, si aggiungono le funzioni ***_cs()***, ***_ds()***, ***_ss()***, ***_es()***, ***_sp()*** e ***_bp()***; le quali, rispettivamente, consentono di leggere il valore dei registri ***CS***, ***DS***, ***SS***, ***ES***, ***SP*** e ***BP*** (le macro-istruzioni equivalenti sono ***cs()***, ***ds()***, ***ss()***, ***es()***, ***sp()*** e ***bp()***).

Tabella 90.25. Funzioni e macro-istruzioni legate strettamente all'hardware, dichiarate nel file di intestazione ‘`lib/sys/os16.h`’. Tali funzioni e macro-istruzioni possono essere utilizzate sia dal kernel, sia dalle applicazioni.

Funzione o macro-istruzione	Descrizione
`uint16_t _seg_i (void);` `unsigned int seg_i (void);`	Restituisce il numero del segmento codice (*instruction*). Listati 101.12 e 101.12.6.
`uint16_t _seg_d (void);` `unsigned int seg_d (void);`	Restituisce il numero del segmento dati. Listati 101.12 e 101.12.5.
`uint16_t _cs (void);` `unsigned int cs (void);`	Restituisce il valore del registro ***CS***. Listati 101.12 e 101.12.2.
`uint16_t _ds (void);` `unsigned int ds (void);`	Restituisce il valore del registro ***DS***. Listati 101.12 e 101.12.3.
`uint16_t _ss (void);` `unsigned int ss (void);`	Restituisce il valore del registro ***SS***. Listati 101.12 e 101.12.8.

Funzione o macro-istruzione	Descrizione
`uint16_t _es (void);` `unsigned int es (void);`	Restituisce il valore del registro ***ES***. Listati 101.12 e 101.12.4.
`uint16_t _sp (void);` `unsigned int sp (void);`	Restituisce il valore del registro ***SP***. Il valore che si ottiene si riferisce allo stato del registro, prima di chiamare la funzione. Listati 101.12 e 101.12.7.
`uint16_t _bp (void);` `unsigned int bp (void);`	Restituisce il valore del registro ***BP***. Il valore che si ottiene si riferisce allo stato del registro, prima di chiamare la funzione. Listati 101.12 e 101.12.1.

90.3.2 Funzioni di basso livello dei file «kernel/ibm_i86/*»

Listato 100.5 e successivi.

Le funzioni con nomi che iniziano per '**_int*nn*...()**', dove ***nn*** è un numero di due cifre, in base sedici, consentono l'accesso all'interruzione ***nn*** del BIOS dal codice in linguaggio C.

Le funzioni con nomi del tipo '**_in_*n*()**' e '**_out_*n*()**' consentono di leggere e di scrivere un valore di ***n*** bit in una certa porta.

La funzione ***cli()*** disabilita le interruzioni hardware, mentre ***sti()*** le riabilita. Queste due funzioni vengono usate pochissimo nel codice del kernel. A loro si aggiungono le funzioni ***irq_on()*** e ***irq_off()***, per abilitare o escludere selettivamente un tipo di interruzione hardware. Queste funzioni vengono usate in una sola occasione, quando si predispone la tabella IVT e poi si abilitano esclusivamente le interruzioni utili.

La funzione ***_ram_copy()*** si occupa di copiare una quantità stabilita di byte da una posizione della memoria a un'altra, entrambe indicate con segmento e scostamento (la funzione ***mem_copy()*** elencata in '`kernel/memory.h`' si avvale in pratica di questa).

Per agevolare l'uso di queste funzioni, senza costringere a convertire i valori numerici, sono disponibili diverse macro-istruzioni con nomi equivalenti, ma privi del trattino basso iniziale.

Tabella 90.26. Funzioni e macro-istruzioni di basso livello, dichiarate nel file di intestazione 'kernel/ibm_i86.h' e descritte nei file della directory 'kernel/ibm_i860/'. Le macro-istruzioni hanno argomenti di tipo numerico non precisato, purché in grado di rappresentare il valore necessario.

Funzione o macro-istruzione	Descrizione
`void _int10_00 (uint16_t` ***video_mode***`);` `void int10_00 (`***video_mode***`);`	Imposta la modalità video della console. Questa funzione viene usata solo da ***con_init()***, per inizializzare la console; la modalità video è stabilita dalla macro-variabile ***IBM_I86_VIDEO_MODE***, dichiarata nel file 'kernel/ibm_i86.h'.

`void _int10_02 (uint16_t` ***page***`,`
`uint16_t` ***position***`);`
`void int10_02 (`***page***`,` ***position***`);`
	Colloca il cursore in una posizione determinata dello schermo, relativo a una certa pagina video.
	Questa funzione viene usata solo da ***con_putc()***.

`void _int10_05 (uint16_t` ***page***`);` `void int10_05 (`***page***`);`	Seleziona la pagina attiva del video. Questa funzione viene usata solo da ***con_init()*** e ***con_select()***.
`void _int12 (void);` `void int12 (void);`	Restituisce la quantità di memoria disponibile, in multipli di 1024 byte.
`void _int13_00 (uint16_t` ***drive***`);` `void int13_00 (`***drive***`);`	Azzera lo stato dell'unità a disco indicata, rappresentata da un numero secondo le convenzioni del BIOS. Viene usata solo dalle funzioni '**dsk_...()**' che si occupano dell'accesso alle unità a disco.

Funzione o macro-istruzione	Descrizione

`uint16_t _int13_02 (uint16_t` ***drive***`,`
`uint16_t` ***sectors***`,`
`uint16_t` ***cylinder***`,`
`uint16_t` ***head***`,`
`uint16_t` ***sector***`,`
`void *`***buffer***`);`
`void int13_02 (`***drive***`,` ***sectors***`,`
cylinder`,` ***head***`,`
sector`,` ***buffer***`);`
	Legge dei settori da un'unità a disco.
	Questa funzione viene usata soltanto da ***dsk_read_sectors()***.

`uint16_t _int13_03 (uint16_t` ***drive***`,`
`uint16_t` ***sectors***`,`
`uint16_t` ***cylinder***`,`
`uint16_t` ***head***`,`
`uint16_t` ***sector***`,`
`void *`***buffer***`);`
`void int13_03 (`***drive***`,` ***sectors***`,`
cylinder`,` ***head***`,`
sector`,` ***buffer***`);`
	Scrive dei settori in un'unità a disco.
	Questa funzione viene usata solo da ***dsk_write_sectors()***.

`uint16_t _int16_00 (void);` `void int16_00 (void);`	Legge un carattere dalla tastiera, rimuovendolo dalla memoria tampone relativa. Viene usata solo in alcune funzioni di controllo della console, denominate '**`con_`**...**`()`**'.
`uint16_t _int16_01 (void);` `void int16_01 (void);`	Verifica se è disponibile un carattere dalla tastiera: se c'è ne restituisce il valore, ma senza rimuoverlo dalla memoria tampone relativa, altrimenti restituisce zero. Viene usata solo dalle funzioni di gestione della console, denominate '**`con_`**...**`()`**'.

Funzione o macro-istruzione	Descrizione

`void _int16_02 (void);`
 	
`void int16_02 (void);`
	Restituisce un valore con cui è possibile determinare quali funzioni speciali della tastiera risultano inserite (inserimento, fissa-maiuscole, blocco numerico, ecc.). Al momento la funzione non viene usata.

`uint16_t _in_8 (uint16_t` ***port***`);`
 	
`void in_8 (`***port***`);`
	Legge un byte dalla porta di I/O indicata. Questa funzione viene usata da ***irq_on()***, ***irq_off()*** e ***dev_mem()***.

`uint16_t _in_16 (uint16_t` ***port***`);`
 	
`void in_16 (`***port***`);`
	Legge un valore a 16 bit dalla porta di I/O indicata. Questa funzione viene usata solo da ***dev_mem()***.

`void _out_8 (uint16_t` ***port***`,`
`uint16_t` ***value***`);`
`void out_8 (`***port***`,` ***value***`);`
	Scrive un byte nella porta di I/O indicata. Questa funzione viene usata da ***irq_on()***, ***irq_off()*** e ***dev_mem()***.

`void _out_16 (uint16_t` ***port***`,`
`uint16_t` ***value***`);`
`void out_16 (`***port***`,` ***value***`);`
	Scrive un valore a 16 bit nella porta indicata. Questa funzione viene usata solo da ***dev_mem()***.

`void cli (void);`	Azzera l'indicatore delle interruzioni, nel registro ***FLAGS***. La funzione serve a permettere l'uso dell'istruzione '**`CLI`**' dal codice in linguaggio C, ma in questa veste, viene usata solo dalla funzione ***proc_init()***.
`void sti (void);`	Attiva l'indicatore delle interruzioni, nel registro ***FLAGS***. La funzione serve a permettere l'uso dell'istruzione '**`STI`**' dal codice in linguaggio C, ma in questa veste, viene usata solo dalla funzione ***proc_init()***.

Funzione o macro-istruzione	Descrizione
`void irq_on (unsigned int` ***irq***`);`	Abilita l'interruzione hardware indicata. Questa funzione viene usata solo da ***proc_init()***.
`void irq_off (unsigned int` ***irq***`);`	Disabilita l'interruzione hardware indicata. Questa funzione viene usata solo da ***proc_init()***.

`void _ram_copy (segment_t` ***org_seg***`,`
`offset_t` ***org_off***`,`
`segment_t` ***dst_seg***`,`
`offset_t` ***dst_off***`,`
`uint16_t` ***size***`);`
`void ram_copy (`***org_seg***`,`
org_off`,`
dst_seg`,`
dst_off`,`
size`);`
	Copia una certa quantità di byte, da una posizione di memoria all'altra, specificando segmento e scostamento di origine e destinazione. Viene usata solo dalle funzioni '**`mem_...()`**'.

90.3.3 Gestione della console

Listato 100.5 e successivi.

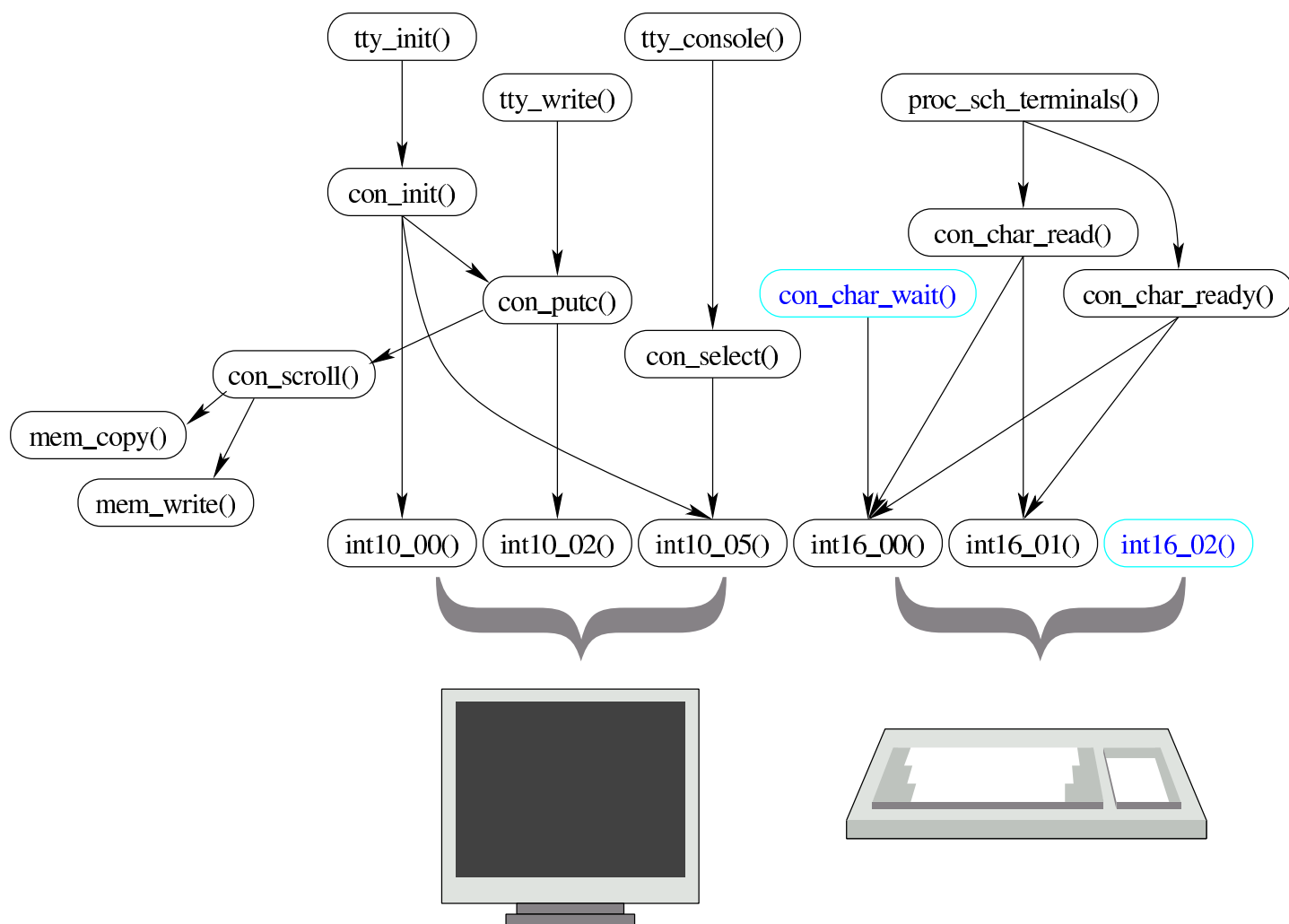
La console offre solo funzionalità elementari, dove è possibile scrivere o leggere un carattere alla volta, sequenzialmente. Ci sono al massimo quattro console virtuali, selezionabili attraverso le combinazioni di tasti [*Ctrl q*], [*Ctrl r*], [*Ctrl s*] e [*Ctrl t*] (ma nella configurazione predefinita vengono attivate solo le prime due) e non è possibile controllare i colori o la posizione del testo che si va a esporre; in pratica si opera come su una telescrivente. Le funzioni di livello più basso, relative alla console hanno nomi che iniziano per '**`con_...()`**'.

Nel codice del kernel si vede usata frequentemente la funzione ***k_printf()***, la quale va a utilizzare la funzione ***k_vprintf()***, dove poi, attraverso altri passaggi, si arriva a utilizzare la funzione ***con_putc()***.

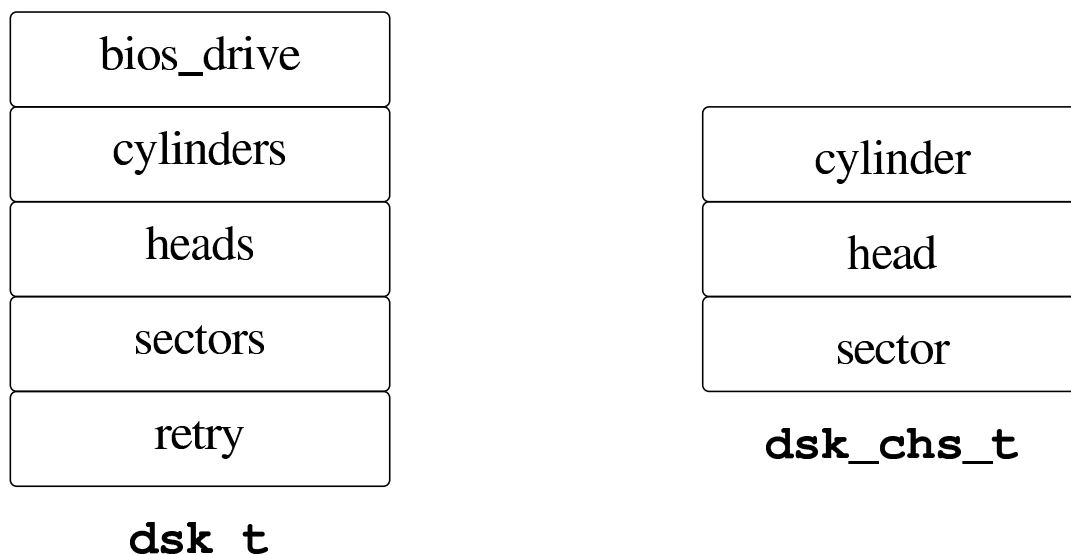
Tabella 90.27. Funzioni per l'accesso alla console, dichiarate nel file di intestazione '`kernel/ibm_i86.h`' e descritte nei file contenuti nella directory '`kernel/ibm_i86/`'.

Funzione	Descrizione
`int con_char_read (void);`	Legge un carattere dalla console, se questo è disponibile, altrimenti restituisce il valore zero. Questa funzione viene usata solo da ***proc_sch_terminals()***.
`int con_char_wait (void);`	Legge un carattere dalla console, ma se questo non è ancora disponibile, rimane in attesa, bloccando tutto il sistema operativo. Questa funzione non è utilizzata.
`int con_char_ready (void);`	Verifica se è disponibile un carattere dalla console: se è così, restituisce un valore diverso da zero, corrispondente al carattere in attesa di essere prelevato. Questa funzione viene usata solo da ***proc_sch_terminals()***.
`void con_init (void);`	Inizializza la gestione della console. Questa funzione viene usata solo da ***tty_init()***.
`void con_select (int` ***console***`);`	Seleziona la console desiderata, dove la prima si individua con lo zero. Questa funzione viene usata solo da ***tty_console()***.
`void con_putc (int` ***console***`, int` ***c***`);`	Visualizza il carattere indicato sullo schermo della console specificata, sulla posizione in cui si trova il cursore, facendolo avanzare di conseguenza e facendo scorrere il testo in alto, se necessario. Questa funzione viene usata solo da ***tty_write()***.
`void con_scroll (int` ***console***`);`	Fa avanzare in alto il testo della console selezionata. Viene usata internamente, solo dalla funzione ***con_putc()***.

Nella figura successiva si vede l'interdipendenza tra le funzioni relative alla gestione di basso livello della console. In un altro capitolo si descrivono le funzioni '**`tty_...()`**', con le quali si gestiscono i terminali in forma più astratta. Nello schema successivo si può vedere che la funzione ***con_scroll()*** si avvale di funzioni per la gestione della memoria: infatti, lo scorrimento del testo dello schermo si ottiene intervenendo direttamente nella memoria utilizzata per la rappresentazione del testo sullo schermo.

Figura 90.28. Interdipendenza tra le funzioni relative alla gestione della console.

90.3.4 Gestione dei dischi

Listato 100.5 e successivi.

Nel file '`ibm_i86.h`' vengono definiti due tipi derivati: '**`dsk_t`**' per annotare le caratteristiche di un disco; '**`dsk_chs_t`**' per annotare simultaneamente le coordinate di accesso a un disco, formate dal numero del cilindro, della testina e del settore.

Le funzioni con nomi del tipo '`dsk_...()`' riguardano l'accesso ai dischi, a livello di settore o di byte, e utilizzano le informazioni annotate nell'array ***dsk_table[]***, composto da elementi di tipo '`dsk_t`'. In pratica, l'array ***dsk_table[]*** viene creato con '`DSK_MAX`' elementi (pertanto solo quattro), uno per ogni disco che si intende gestire. Quando le funzioni '`dsk_...()`' richiedono l'indicazione di un numero di unità (*drive*), si riferiscono all'indice dell'array ***dsk_table[]*** (al contrario, le funzioni '`_int13_...()`' hanno come riferimento il codice usato dal BIOS).

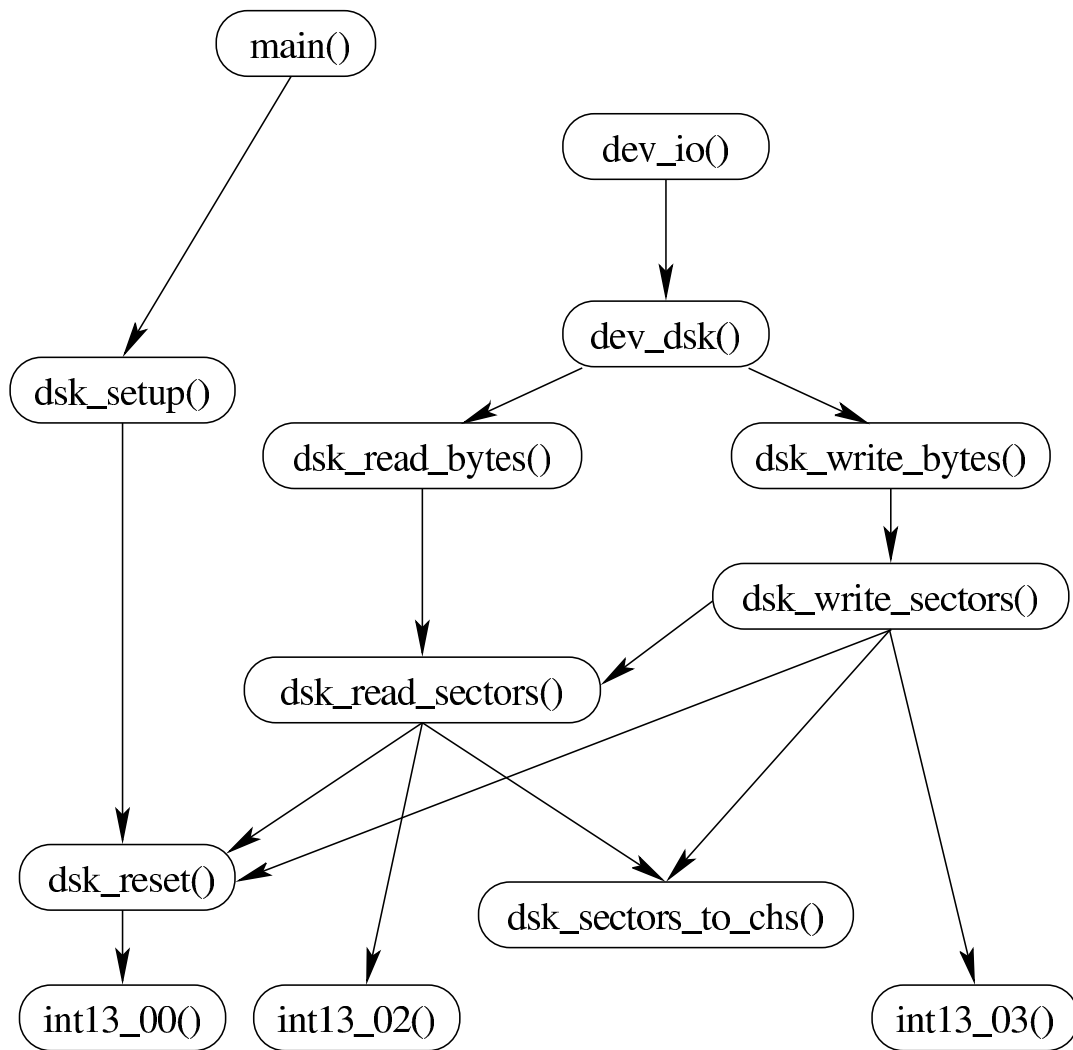
La funzione ***dsk_setup()*** compila l'array ***dsk_table[]*** con i dati relativi ai dischi che si utilizzano; la funzione ***dsk_reset()*** azzera la funzionalità di una certa unità; la funzione ***dsk_sector_to_chs()*** converte il numero assoluto di un settore nelle coordinate corrispondenti (cilindro, testina e settore).

Le funzioni ***dsk_read_sectors()*** e ***dsk_write_sectors()*** servono a leggere o scrivere una quantità stabilita di settori, usando come appoggio un'area di memoria individuata da un puntatore generico. Le funzioni ***dsk_read_bytes()*** e ***dsk_write_bytes()*** svolgono un compito equivalente, ma usando come riferimento il byte; in questo caso, restituiscono la quantità di byte letti o scritti rispettivamente.

Tabella 90.30. Funzioni per l'accesso ai dischi, dichiarate nel file di intestazione '`kernel/ibm_i86.h`'.

Funzione	Descrizione
`void dsk_setup (void);`	Predispone il contenuto dell'array ***dsk_table[]***. Questa funzione viene usata soltanto da ***main()***.
`int dsk_reset (int` ***drive***`);`	Azzera lo stato dell'unità corrispondente a ***dsk_table[drive].bios_drive***. Viene usata solo internamente, dalle altre funzioni '`dsk_...()`'.
`void dsk_sector_to_chs (int` ***drive***`, unsigned int` ***sector***`, dsk_chs_t *`***chs***`);`	Modifica le coordinate della variabile strutturata a cui punta l'ultimo parametro, con le coordinate corrispondenti al numero di settore fornito. Viene usata solo internamente, dalle altre funzioni '`dsk_...()`'.

Funzione	Descrizione
`int dsk_read_sectors (int` ***drive***`, unsigned int` ***start_sector***`, void *`***buffer***`, unsigned int` ***n_sectors***`);`	Legge una sequenza di settori da un disco, mettendo i dati in memoria, a partire dalla posizione espressa da un puntatore generico. La funzione è ricorsiva, ma oltre che da se stessa, viene usata internamente da ***dsk_read_bytes()*** e da ***dsk_write_bytes()***.
`int dsk_write_sectors (int` ***drive***`, unsigned int` ***start_sector***`, void *`***buffer***`, unsigned int` ***n_sectors***`);`	Scrive una sequenza di settori in un disco, traendo i dati da un puntatore a una certa posizione della memoria. La funzione è ricorsiva, ma oltre che da se stessa, viene usata solo internamente da ***dsk_write_bytes()***.
`size_t dsk_read_bytes (int` ***drive***`, off_t` ***offset***`, void *`***buffer***`, size_t count);`	Legge da una certa unità a disco una quantità specificata di byte, a partire dallo scostamento indicato (nel disco), il quale deve essere un valore positivo. Questa funzione viene usata solo da ***dev_dsk()***.
`size_t dsk_write_bytes (int` ***drive***`, off_t` ***offset***`, void *`***buffer***`, size_t count);`	Scrive su una certa unità a disco una quantità specificata di byte, a partire dallo scostamento indicato (nel disco), il quale deve essere un valore positivo. Questa funzione viene usata solo da ***dev_dsk()***.

Figura 90.31. Interdipendenza tra le funzioni relative all'accesso ai dischi.

90.4 Gestione della memoria

Dal punto di vista del kernel di os16, l'allocazione della memoria riguarda la collocazione dei processi elaborativi nella stessa. Disponendo di una quantità di memoria esigua, si utilizza una mappa di bit per indicare lo stato dei blocchi di memoria, dove un bit a uno indica un blocco di memoria occupato.

Nel file '`memory.h`' viene definita la dimensione di un blocco di memoria e, di conseguenza, la quantità massima che possa essere gestita. Attualmente i blocchi sono da 256 byte, pertanto, sapendo che la memoria può arrivare solo fino a 640 Kibyte, si gestiscono al massimo 2560 blocchi.

Per la scansione della mappa si utilizzano interi da 16 bit, pertanto tutta la mappa si riduce a 40 di questi interi, ovvero 80 byte. Nell'ambito di ogni intero da 16 bit, il bit più significativo rappresenta il primo blocco di memoria di sua competenza. Per esempio, per indicare che si stanno utilizzando i primi 1280 byte, pari ai primi cinque blocchi di memoria, si rappresenta la mappa della memoria come «F80000000...».

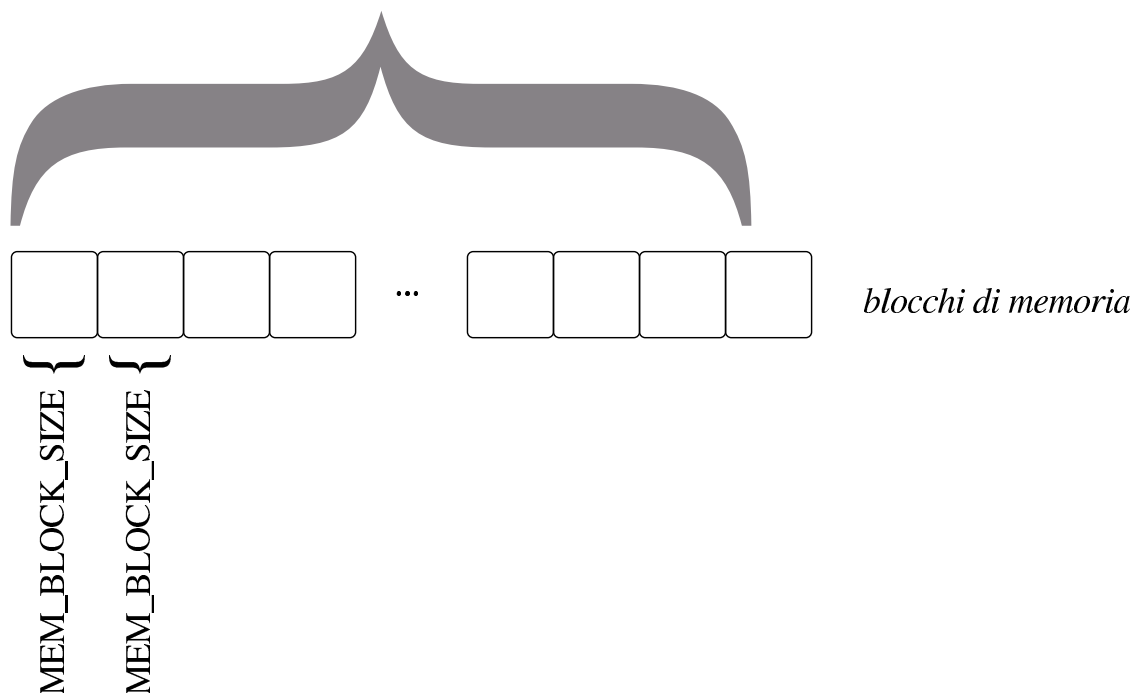
Il fatto che la mappa della memoria vada scandito a ranghi di 16 bit va tenuto in considerazione, perché se invece si andasse con ranghi differenti, si incapperebbe nel problema dell'inversione dei byte.

Quando possibile, si fa riferimento a indirizzi di memoria efficaci, nel senso che, con un solo valore, si rappresentano le posizioni da 00000_{16} a $FFFFF_{16}$. Per questo viene predisposto il tipo derivato '**`addr_t`**' nel file '`memory.h`'.

90.4.1 File «kernel/memory.h» e «kernel/memory/...»

Listato 100.8 e successivi.

Il file '`kernel/memory.h`', oltre ai prototipi delle funzioni usate per la gestione della memoria, definisce la dimensione del blocco minimo di memoria e la quantità massima di questi, rispettivamente con le macro-variabili ***MEM_BLOCK_SIZE*** e ***MEM_MAX_BLOCKS***; inoltre predispone i tipi derivati '**`memory_t`**', '**`segment_t`**', '**`offset_t`**' e '**`addr_t`**', corrispondenti, rispettivamente, a una variabile strutturata che consente di rappresentare un'area di memoria in modo relativamente comodo (indirizzo efficace, segmento e dimensione), un indirizzo di segmento, uno scostamento dall'inizio di un segmento e un indirizzo efficace.

Figura 90.32. Mappa della memoria in blocchi: la dimensione minima di un'area di memoria è di '**`MEM_BLOCK_SIZE`**' byte.

Nei file della directory '`kernel/memory/`' viene dichiarata la mappa della memoria, corrispondente a un array di interi a 16 bit, denominato ***mb_table[]***. L'array è pubblico, tuttavia è disponibile anche una funzione che ne restituisce il punta-

tore: ***mb_reference()***. Tale funzione sarebbe perfettamente inutile, ma rimane per uniformità rispetto alla gestione delle altre tabelle.

Nelle funzioni che riguardano l'allocazione della memoria, quando si indica la dimensione di questa, spesso si considera il valore zero equivalente a 10000_{16}, ovvero la dimensione massima di un segmento secondo l'architettura.

Tabella 90.33. Funzioni per la gestione della mappa della memoria, dichiarate nel file di intestazione '`kernel/memory.h`' e realizzate nella directory '`kernel/memory/`'.

Funzione	Descrizione
`uint16_t *mb_reference (void);`	Restituisce il puntatore alla tabella dei blocchi di memoria, per uniformare l'accesso alla tabella dalle funzioni che non fanno parte del gruppo contenuto nella directory '`kernel/memory/`'.
`ssize_t mb_alloc (addr_t` ***address***`,` `size_t` ***size***`);`	Alloca la memoria a partire dall'indirizzo efficace indicato, per la quantità di byte richiesta (zero corrisponde a 10000_{16} byte). L'allocazione ha termine anticipatamente se si incontra un blocco già utilizzato. La funzione restituisce la dimensione allocata effettivamente.
`ssize_t mb_free (addr_t` ***address***`,` `size_t` ***size***`);`	Libera la memoria a partire dall'indirizzo efficace indicato, per la quantità di byte richiesta (zero corrisponde a 10000_{16} byte). Lo spazio viene liberato in ogni caso, anche se risulta già libero; tuttavia viene prodotto un avvertimento a video se si verifica tale ipotesi.

Funzione	Descrizione
`int mb_alloc_size (size_t` ***size***`, memory_t *`***allocated***`);`	Cerca e alloca un'area di memoria della dimensione richiesta, modificando la variabile strutturata di cui viene fornito il puntatore come secondo parametro. In pratica, l'indirizzo e l'estensione della memoria allocata effettivamente si trovano nella variabile strutturata in questione, mentre la funzione restituisce zero (se va tutto bene) o −1 se non è disponibile la memoria libera richiesta.

Tabella 90.34. Funzioni per le operazioni di lettura e scrittura in memoria, dichiarate nel file di intestazione '`kernel/memory.h`' e realizzate nella directory '`kernel/memory/`'.

Funzione	Descrizione
`void mem_copy (addr_t` ***orig***`, addr_t` ***dest***`, size_t` ***size***`);`	Copia la quantità richiesta di byte, dall'indirizzo di origine a quello di destinazione, espressi in modo efficace.
`size_t mem_read (addr_t` ***start***`, void *`***buffer***`, size_t` ***size***`);`	Legge dalla memoria, a partire dall'indirizzo indicato come primo parametro, la quantità di byte indicata come ultimo parametro. Ciò che viene letto va poi copiato nella memoria tampone corrispondente al puntatore generico indicato come secondo parametro.
`size_t mem_write (addr_t` ***start***`, void *`***buffer***`, size_t` ***size***`);`	Scrive, in memoria, a partire dall'indirizzo indicato come primo parametro, la quantità di byte indicata come ultimo parametro. Ciò che viene scritto proviene dalla memoria tampone corrispondente al puntatore generico indicato come secondo parametro.

90.4.2 Scansione della mappa di memoria

«

Listato 100.8 e successivi.

La mappa della memoria si rappresenta (a sua volta in memoria), con un array di interi a 16 bit, dove ogni bit individua un blocco di memoria. Pertanto, l'array si compone di una quantità di elementi pari al valore di '**MEM_MAX_BLOCKS**' diviso 16.

Il primo elemento di questo array, ovvero ***mb_table[0]***, individua i primi 16 blocchi di memoria, dove il bit più significativo si riferisce precisamente al primo blocco. Per esempio, se ***mb_table[0]*** contiene il valore $F800_{16}$, ovvero 1111100000000000_{2}, significa che i primi cinque blocchi di memoria sono occupati, mentre i blocchi dal sesto al sedicesimo sono liberi.

Dal momento che i calcoli per individuare i blocchi di memoria e per intervenire nella mappa relativa, possono creare confusione, queste operazioni sono raccolte in funzioni statiche separate, anche se sono utili esclusivamente all'interno del file in cui si trovano. Tali funzioni statiche hanno una sintassi comune:

```
int mb_block_set1   (int block)
int mb_block_set0   (int block)
int mb_block_status (int block)
```

Le funzioni ***mb_block_set1()*** e ***mb_block_set0()*** servono rispettivamente a impegnare o liberare un certo blocco di memoria, individuato dal valore dell'argomento. La funzione ***mb_block_status()*** restituisce uno nel caso il blocco indicato risulti allocato, oppure zero in caso contrario.

Queste tre funzioni usano un metodo comune per scandire la mappa della memoria: il valore che rappresenta il blocco a cui si vuole fare riferimento, viene diviso per 16, ovvero il rango degli elementi dell'array che rappresenta la mappa della memoria. Il risultato intero della divisione serve per trovare quale elemento dell'array considerare, mentre il resto della divisione serve per determinare quale bit dell'elemento trovato rappresenta il blocco desiderato. Trovato ciò, si deve costruire una maschera, nella quale si mette a uno il bit che rappresenta il blocco; per farlo, si pone inizialmente a uno il bit più significativo della maschera, quindi lo si fa scorrere verso destra di un valore pari al resto della divisione.

Per esempio, volendo individuare il terzo blocco di memoria, pari al numero 2 (il primo blocco corrisponderebbe allo zero), si avrebbe che questo è descritto dal primo elemento dell'array (in quanto 2/16 dà zero, come risultato intero), mentre la maschera necessaria a trovare il bit corrispondente è 0010000000000000_{2}, la quale si ottiene

spostando per due volte verso destra il bit più significativo (due volte, pari al resto della divisione).

Una volta determinata la maschera, per segnare come occupato un blocco di memoria, basta utilizzare l'operatore OR binario:

```
        mb_table[i] = mb_table[i] | mask;
```

Se invece si vuole liberare un blocco di memoria, si utilizza un AND binario, invertendo però il contenuto della maschera:

```
        mb_table[i] = mb_table[i] & ~mask;
```

Va osservato che la rappresentazione dei blocchi nella mappa è invertita rispetto ad altri sistemi operativi, in quanto non sarebbe tanto logico il fatto che il bit più significativo si riferisca invece alla parte più bassa del proprio insieme di blocchi di memoria. La scelta è dovuta al fatto che, volendo rappresentare la mappa numericamente, la lettura di questa sarebbe più vicina a quella che è la percezione umana del problema.

90.5 Gestione dei terminali virtuali

Listato 100.10 e successivi.

os16 offre esclusivamente un utilizzo operativo tramite console. Alcune funzioni con prefisso '**con_...()**', dichiarate nel file '`kernel/ibm_i86.h`', si occupano di tale gestione, ma per distinguere tra terminali virtuali (o console virtuali), associati a processi differenti, si rende necessario un livello ulteriore di astrazione, costituito dal codice contenuto nel file '`kernel/tty.h`' e in quelli della directory '`kernel/tty/`'.

I terminali virtuali gestibili sono rappresentati da un array di variabili strutturate, ognuna delle quali contiene tutte le informazioni del contesto operativo di un certo terminale. L'array in questione è ***tty_table[]*** (a cui però si accede tramite una funzione che ne restituisce il puntatore) e vi si annotano, per ogni terminale attivo, il numero del dispositivo corrispondente, il numero del gruppo di processi a cui si associa, l'ultimo codice digitato (e non ancora letto), lo stato di funzionamento (se sono stati persi dei dati o meno).

Va osservata la differenza sostanziale che c'è tra le operazioni di scrittura e quelle di lettura. Infatti, la scrittura sul terminale implica la chiamata della funzione ***con_putc()***, del file '`kernel/ibm_i86.h`'; al contrario, la lettura avviene in forma passiva, limitandosi ad acquisire il valore già disponibile nella variabile strutturata che rappresenta il terminale virtuale. Come può essere verificato successivamente, è compito del sistema di gestione delle interruzioni la fornitura del valore digitato

al terminale virtuale competente, tramite l'appoggio della variabile strutturata che lo rappresenta.

Come per tutte le tabelle di os16 che non fanno parte di uno standard, anche quella che contiene le informazioni dei terminali virtuali è accessibile preferibilmente con l'ausilio di una funzione che ne restituisce il puntatore. In questo caso, la funzione ***tty_reference()*** consente di ottenere il puntatore all'elemento corrispondente della tabella dei terminali, fornendo come argomento il numero del dispositivo cercato.

Tabella 90.37. Funzioni per la gestione dei terminali, dichiarate nel file di intestazione '`kernel/tty.h`'.

Funzione	Descrizione
`void tty_init (void);`	Inizializza la gestione dei terminali. Viene usata una volta sola nella funzione ***main()*** del kernel.
`tty_t *tty_reference (dev_t` ***device***`);`	Restituisce il puntatore a un elemento della tabella dei terminali. Se come numero di dispositivo si indica lo zero, si ottiene il riferimento a tutta la tabella; se non viene trovato il numero di dispositivo cercato, si ottiene il puntatore nullo.
`dev_t tty_console (dev_t` ***device***`);`	Seleziona la console indicata attraverso il numero di dispositivo che costituisce l'unico parametro. Se viene dato un valore a zero, si ottiene solo di conoscere qual è la console attiva. La console selezionata viene anche memorizzata in una variabile statica, per le chiamate successive della funzione. Se viene indicato un numero di dispositivo non valido, si seleziona implicitamente la prima console.

Funzione	Descrizione
`int tty_read (dev_t` ***device***`);`	Legge un carattere dal terminale specificato attraverso il numero di dispositivo. Per la precisione, il carattere viene tratto dal campo relativo contenuto nella tabella dei terminali. Il carattere viene restituito dalla funzione come valore intero comune; se si ottiene zero significa che non è disponibile alcun carattere.
`void tty_write (dev_t` ***device***`, int` ***c***`);`	Scrive sullo schermo del terminale rappresentato dal numero di dispositivo, il carattere fornito come secondo parametro.

90.6 Dispositivi

La gestione dei dispositivi fisici, da parte di os16, è molto limitata, in quanto ci si avvale prevalentemente delle funzionalità già offerte dal BIOS. In ogni caso, tutte le operazioni di lettura e scrittura di dispositivi, passano attraverso la gestione comune della funzione ***dev_io()***.

90.6.1 File «lib/sys/os16.h» e directory «lib/sys/os16/»

Listato 101.12 e altri.

Una parte delle definizioni che riguardano la gestione dei dispositivi, necessaria al kernel, è disponibile anche alle applicazioni attraverso il file '`lib/sys/os16.h`'. Al suo interno, tra le altre cose, è definito un insieme di macro-variabili, con prefisso ***DEV_***..., con cui si distinguono i numeri attribuiti ai dispositivi. Per esempio, ***DEV_DSK_MAJOR*** corrisponde al numero primario (*major*) per tutte le unità di memorizzazione, mentre ***DEV_DSK1*** corrisponde al numero primario e secondario (*major* e *minor*), in un valore unico, della seconda unità a disco.

90.6.2 File «kernel/devices.h» e «kernel/devices/...»

Listati 100.2 e altri.

Il file '`kernel/devices.h`' incorpora il file '`lib/sys/os16/os16.h`', per acquisire le funzionalità legate alla gestione dei dispositivi che sono disponibili anche agli applicativi. Successivamente dichiara la funzione ***dev_io()***, la quale sintetizza tutta la gestione dei dispositivi. Questa funzione utilizza il parametro ***rw***, per specificare l'azione da svolgere (lettura o scrittura). Per questo parametro vanno usate

le macro-variabili ***DEV_READ*** e ***DEV_WRITE***, così da non dover ricordare quale valore numerico corrisponde alla lettura e quale alla scrittura.

```
ssize_t dev_io (pid_t pid, dev_t device, int rw, off_t offset,
                void *buffer, size_t size, int *eof);
```

Sono comunque descritte anche altre funzioni, ma utilizzate esclusivamente da ***dev_io()***.

La funzione ***dev_io()*** si limita a estrapolare il numero primario dal numero del dispositivo complessivo, quindi lo confronta con i vari tipi gestibili. A seconda del numero primario seleziona una funzione appropriata per la gestione di quel tipo di dispositivo, passando praticamente gli stessi argomenti già ricevuti.

Va osservato il caso particolare dei dispositivi '**DEV_KMEM_**...'. In un sistema operativo Unix comune, attraverso ciò che fa capo al file di dispositivo '`/dev/kmem`', si ha la possibilità di accedere all'immagine in memoria del kernel, lasciando a un programma con privilegi adeguati la facoltà di interpretare i simboli che consentono di individuare i dati esistenti. Nel caso di os16, non ci sono simboli nel risultato della compilazione, quindi non è possibile ricostruire la collocazione dei dati. Per questa ragione, le informazioni che devono essere pubblicate, vengono controllate attraverso un dispositivo specifico. Quindi, il dispositivo '**DEV_KMEM_PS**' consente di leggere la tabella dei processi, '**DEV_KMEM_MMAP**' consente di leggere la mappa della memoria, e così vale anche per altre tabelle.

Per quanto riguarda la gestione dei terminali, attraverso la funzione ***dev_tty()***, quando un processo vuole leggere dal terminale, ma non risulta disponibile un carattere, questo viene messo in pausa, in attesa di un evento legato ai terminali.

os16, non disponendo di un sistema di trattenimento dei dati in memoria (*cache*), esegue le operazioni di lettura e scrittura dei dispositivi in modo immediato. Per questo motivo, la distinzione tra file di dispositivo a blocchi e a caratteri, rimane puramente estetica, ovvero priva di un'utilità concreta.

Figura 90.38. Interdipendenza tra la funzione ***dev_io()*** e le altre. I collegamenti con le funzioni ***major()*** e ***minor()*** sono omesse.

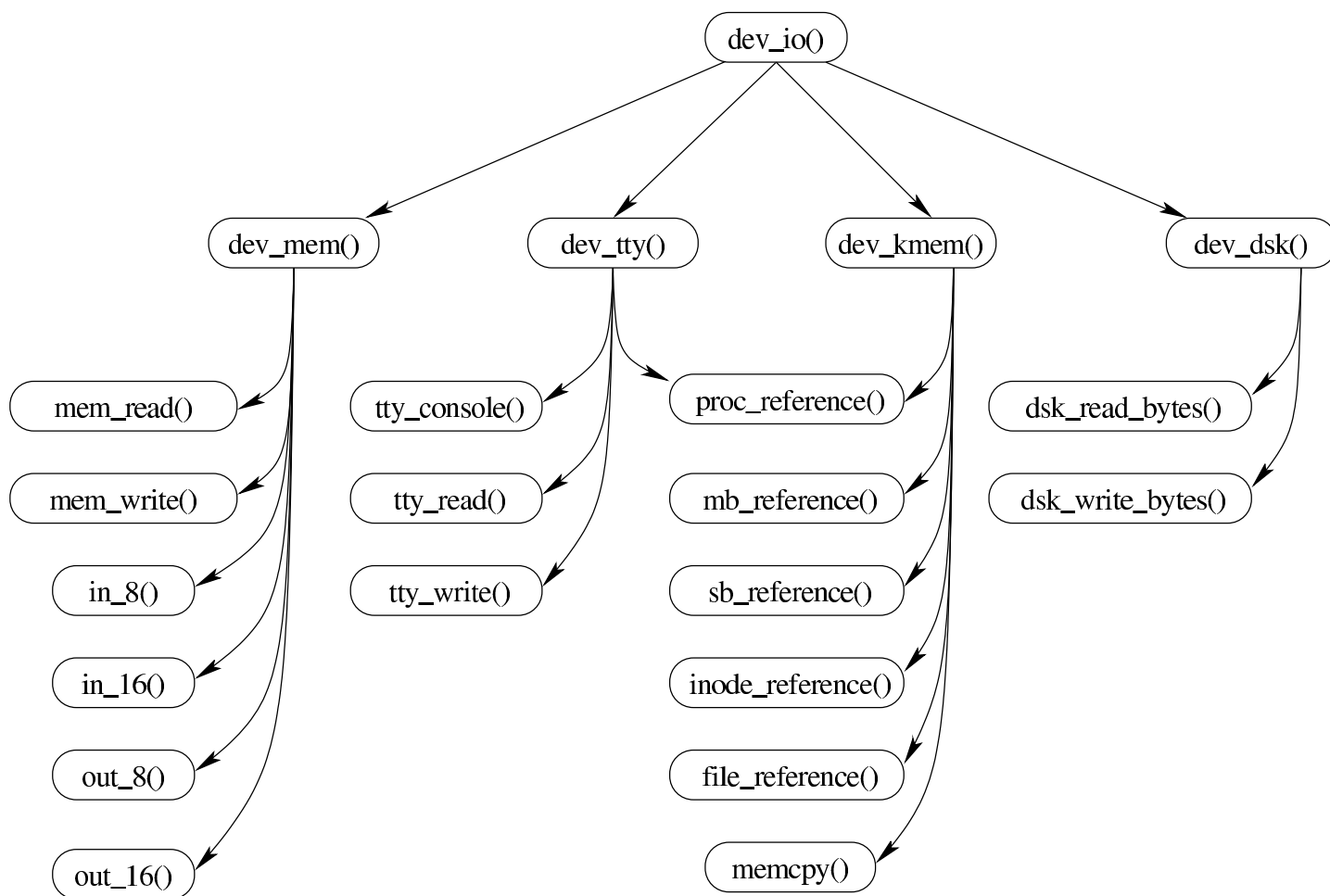
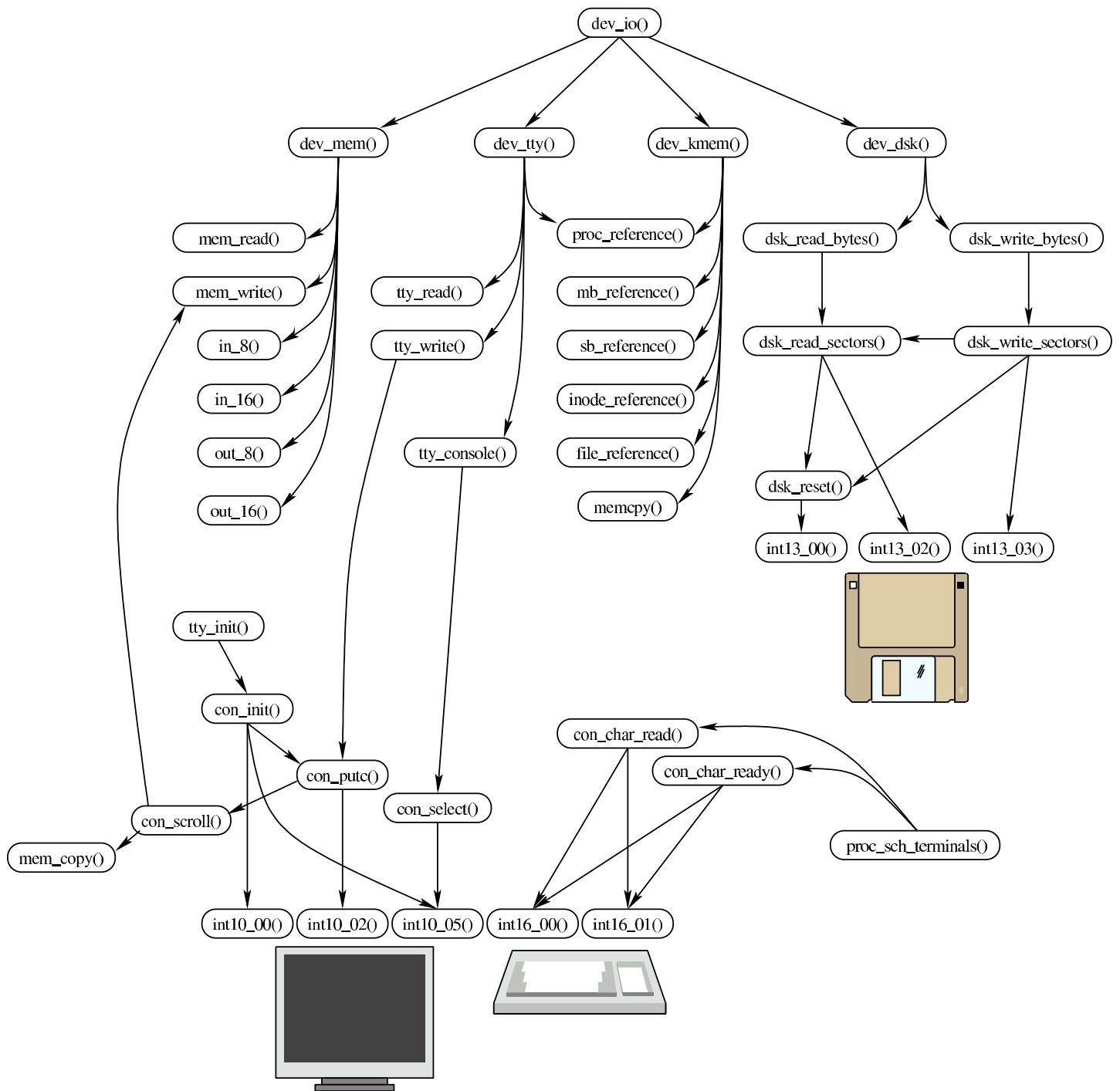



Figura 90.39. Schema più ampio delle dipendenze che hanno origine dalla funzione ***dev_io()***.

90.6.3 Numero primario e numero secondario

«

I dispositivi, secondo la tradizione dei sistemi Unix, sono rappresentati dal punto di vista logico attraverso un numero intero, senza segno, a 16 bit. Tuttavia, per organizzare questa numerazione in modo ordinato, tale numero viene diviso in due parti: la prima parte, nota come *major*, ovvero «numero primario», si utilizza per individuare il tipo di dispositivo; la seconda, nota come *minor*, ovvero «numero secondario», si utilizza per individuare precisamente il dispositivo, nell'ambito del tipo a cui appartiene.

In pratica, il numero complessivo a 16 bit si divide in due, dove gli 8 bit più significativi individuano il numero primario, mentre quelli meno significativi danno il numero secondario. L'esempio seguente si riferisce al dispositivo che genera il valore zero, il quale appartiene al gruppo dei dispositivi relativi alla memoria:

`DEV_MEM_MAJOR`	01_{16}
`DEV_ZERO`	0104_{16}

In questo caso, il valore che rappresenta complessivamente il dispositivo è 0104_{16} (pari a 260_{10}), ma si compone di numero primario 01_{16} e di numero secondario 04_{16} (che coincidono nella rappresentazione in base dieci). Per estrarre il numero primario si deve dividere il numero complessivo per 256 (0100_{16}), trattenendo soltanto il risultato intero; per filtrare il numero secondario si può fare la stessa divisione, ma trattenendo soltanto il resto della stessa. Al contrario, per produrre il numero del dispositivo, partendo dai numeri primario e secondario separati, occorre moltiplicare il numero primario per 256, sommando poi il risultato al numero secondario.

90.6.4 Dispositivi previsti

L'astrazione della gestione dei dispositivi, consente di trattare tutti i componenti che hanno a che fare con ingresso e uscita di dati, in modo sostanzialmente omogeneo; tuttavia, le caratteristiche effettive di tali componenti può comportare delle limitazioni o delle peculiarità. Ci sono alcune questioni fondamentali da considerare: un tipo di dispositivo potrebbe consentire l'accesso in un solo verso (lettura o scrittura); l'accesso al dispositivo potrebbe essere ammesso solo in modo sequenziale, rendendo inutile l'indicazione di un indirizzo; la dimensione dell'informazione da trasferire potrebbe assumere un significato differente rispetto a quello comune.

Tabella 90.41. Classificazione dei dispositivi di os16.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_MEM`	r/w	diretto	Permette l'accesso alla memoria, in modo indiscriminato, perché os16 non offre alcun tipo di protezione al riguardo.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_NULL`	r/w	nessuno	Consente la lettura e la scrittura, ma non si legge e non si scrive alcunché.
`DEV_PORT`	r/w	sequenziale	Consente di leggere e scrivere da o verso una porta di I/O, individuata attraverso l'indirizzo di accesso (l'indirizzo, o meglio lo scostamento, viene trattato come la porta a cui si vuole accedere). Tuttavia, la dimensione dell'informazione da trasferire è valida solo se si tratta di uno o di due byte: per la dimensione di un byte si usano le funzioni ***in_8()*** e ***out_8()***; per due byte si usano le funzioni ***in_16()*** e ***out_16()***. Per dimensioni differenti la lettura o la scrittura non ha effetto.
`DEV_ZERO`	r	sequenziale	Consente solo la lettura di valori a zero (zero inteso in senso binario).
`DEV_TTY`	r/w	sequenziale	Rappresenta il terminale virtuale del processo attivo.
`DEV_DSK`***n***	r/w	diretto	Rappresenta l'unità a dischi ***n***. os16 non gestisce le partizioni.
`DEV_KMEM_PS`	r	diretto	Rappresenta la tabella contenente le informazioni sui processi. L'indirizzo di accesso indica il numero del processo di partenza; la dimensione da leggere dovrebbe essere abbastanza grande da contenere un processo, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_KMEM_MMP`	r	sequenziale	Rappresenta la mappa della memoria, alla quale si può accedere solo dal suo principio. In pratica, l'indirizzo di accesso viene ignorato, mentre conta solo la quantità di byte richiesta.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_KMEM_SB`	r	diretto	Rappresenta la tabella dei super blocchi (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il super blocco; la dimensione richiesta dovrebbe essere abbastanza grande da contenere un super blocco, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_KMEM_INODE`	r	diretto	Rappresenta la tabella degli inode (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare l'inode; la dimensione richiesta dovrebbe essere abbastanza grande da contenere un inode, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_KMEM_FILE`	r	diretto	Rappresenta la tabella dei file (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il file; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di un file, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_CONSOLE`	r/w	sequenziale	Legge o scrive relativamente alla console attiva la quantità di byte richiesta, ignorando l'indirizzo di accesso.
`DEV_CONSOLE`***n***	r/w	sequenziale	Legge o scrive relativamente alla console ***n*** la quantità di byte richiesta, ignorando l'indirizzo di accesso.

90.7 Gestione del file system

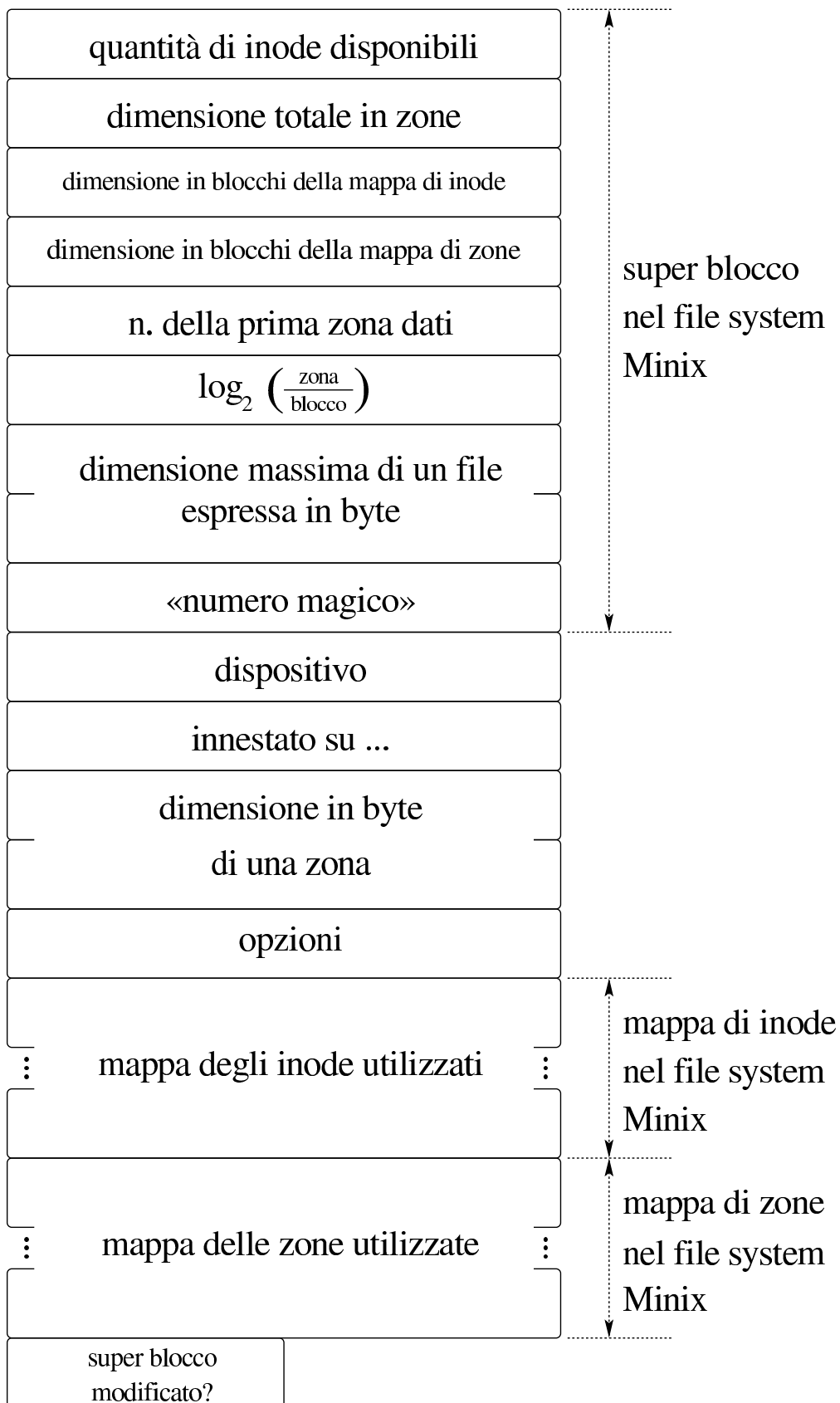
«

La gestione del file system è suddivisa in diversi file contenuti nella directory '`kernel/fs/`', facenti capo al file di intestazione '`kernel/fs.h`'.

90.7.1 File «kernel/fs/sb_...»

«

I file '`kernel/fs/sb_...`' descrivono le funzioni per la gestione dei super blocchi, distinguibili perché iniziano tutte con il prefisso '**`sb_`**'. Tra questi file si dichiara l'array ***sb_table[]***, il quale rappresenta una tabella le cui righe sono rappresentate da elementi di tipo '**`sb_t`**' (il tipo '**`sb_t`**' è definito nel file '`kernel/fs.h`'). Per uniformare l'accesso alla tabella, la funzione ***sb_reference()*** permette di ottenere il puntatore a un elemento dell'array ***sb_table[]***, specificando il numero del dispositivo cercato.

Figura 90.42. Struttura del tipo '`sb_t`', corrispondente agli elementi dell'array ***sb_table[]***.

Listato 90.43. Struttura del tipo '**sb_t**', corrispondente agli elementi dell'array ***sb_table[]***.

```
typedef struct sb       sb_t;

struct sb {
  uint16_t  inodes;
  uint16_t  zones;
  uint16_t  map_inode_blocks;
  uint16_t  map_zone_blocks;
  uint16_t  first_data_zone;
  uint16_t  log2_size_zone;
  uint32_t  max_file_size;
  uint16_t  magic_number;
  //-------------------------
  dev_t     device;
  inode_t  *inode_mounted_on;
  blksize_t blksize;
  int       options;
  uint16_t  map_inode[SB_MAP_INODE_SIZE];
  uint16_t  map_zone[SB_MAP_ZONE_SIZE];
  char      changed;
};
```

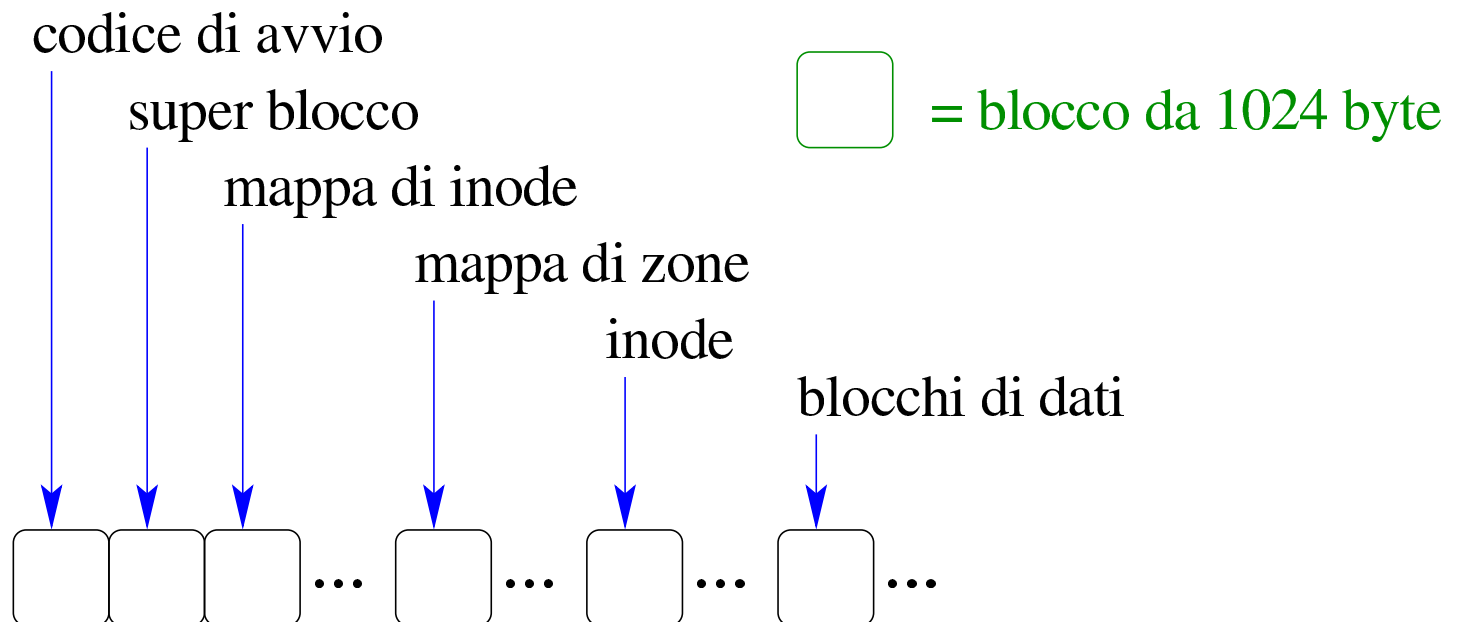
Il super blocco rappresentato dal tipo '**sb_t**' include anche le mappe delle zone e degli inode impegnati. Queste mappe hanno una dimensione fissa in memoria, mentre nel file system reale possono essere di dimensione minore. La tabella di super blocchi, contiene le informazioni dei dispositivi di memorizzazione innestati nel sistema. L'innesto si concretizza nel riferimento a un inode, contenuto nella tabella degli inode (descritta in un altro capitolo), il quale rappresenta la directory di un'altra unità, su cui tale innesto è avvenuto. Naturalmente, l'innesto del file system principale rappresenta un caso particolare.

Tabella 90.44. Funzioni per la gestione dei dispositivi di memorizzazione di massa, a livello di super blocco, definite nei file '`kernel/fs/sb_...`'.

Funzione	Descrizione
`sb_t *sb_reference (dev_t` ***device***`);`	Restituisce il riferimento a un elemento della tabella dei super blocchi, in base al numero del dispositivo di memorizzazione. Se il dispositivo cercato non risulta giù innestato, si ottiene il puntatore nullo; se si chiede il dispositivo zero, si ottiene il puntatore al primo elemento della tabella.
`sb_t *sb_mount (dev_t` ***device***`, inode_t **`***inode_mnt***`, int` ***options***`);`	Innesta il dispositivo rappresentato numericamente dal primo parametro, sulla directory corrispondente all'inode a cui punta il secondo parametro, con le opzioni del terzo parametro. Quando si tratta del primo innesto del file system principale, la directory è quella dello stesso file system, pertanto, in tal caso, ***inode_mnt*** è inizialmente un puntatore nullo e deve essere modificato dalla funzione stessa.
`int sb_save (sb_t *`***sb***`);`	Salva il super blocco nella sua unità di memorizzazione, se questo risulta modificato. In questo caso, il super blocco include anche le mappe degli inode e delle zone.
`int sb_zone_status (sb_t *`***sb***`, zno_t` ***zone***`);`	Restituisce uno se la zona rappresentata dal secondo parametro è impegnata nel super blocco a cui si riferisce il primo parametro; diversamente restituisce zero.

Funzione	Descrizione
`int sb_inode_status (sb_t *sb, ino_t ino);`	Restituisce uno se l'inode rappresentato dal secondo parametro è impegnato nel super blocco a cui si riferisce il primo parametro; diversamente restituisce zero.

90.7.2 File «kernel/fs/zone_...»

Nel file system Minix 1, si distinguono i concetti di blocco e zona di dati, con il vincolo che la zona ha una dimensione multipla del blocco. Il contenuto del file system, dopo tutte le informazioni amministrative, è organizzato in zone; in altri termini, i blocchi di dati si raggiungono in qualità di zone.

La zona rimane comunque un tipo di blocco, potenzialmente più grande (ma sempre multiplo) del blocco vero e proprio, che si numera a partire dall'inizio dello spazio disponibile, con la differenza che è utile solo per raggiungere i blocchi di dati. Nel super blocco del file system si trova l'informazione del numero della prima zona che contiene dati, in modo da non dover ricalcolare questa informazione ogni volta.

I file '`kernel/fs/zone_...`' descrivono le funzioni per la gestione del file system a zone.

Tabella 90.46. Funzioni per la gestione delle zone, definite nei file `kernel/fs/zone_...`.

Funzione	Descrizione
`zno_t zone_alloc (sb_t *sb);`	Alloca una zona, restituendo il numero della stessa. In pratica, cerca la prima zona libera nel file system a cui si riferisce il super blocco ***sb** e la segna come impegnata, restituendone il numero.
`int zone_free (sb_t *sb, zno_t zone);`	Libera una zona, impegnata precedentemente.
`int zone_read (sb_t *sb, zno_t zone, void *buffer);`	Legge il contenuto di una zona, memorizzandolo a partire dalla posizione di memoria rappresentato da ***buffer***.
`int zone_write (sb_t *sb, zno_t zone, void *buffer);`	Sovrascrive una zona, utilizzando il contenuto della memoria a partire dalla posizione rappresentata da ***buffer***.

90.7.3 File «kernel/fs/inode_...»

I file `kernel/fs/inode_...` descrivono le funzioni per la gestione dei file, in forma di inode. In uno di questi file viene dichiarata la tabella degli inode in uso nel sistema, rappresentata dall'array ***inode_table[]*** e per individuare un certo elemento dell'array si usa preferibilmente la funzione ***inode_reference()***. Gli elementi della tabella degli inode sono di tipo **`inode_t`** (definito nel file `kernel/fs.h`); una voce della tabella rappresenta un inode utilizzato se il campo dei riferimenti (***references***) ha un valore maggiore di zero.

Figura 90.47. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.

tipo di file e permessi di accesso

UID: utente proprietario

dimensione del file in byte

data e orario dell'ultima modifica espressa in secondi dal 1/1/1970

n. collegamenti dalle directory

GID: gruppo proprietario

numero della zona 0

numero della zona 1

numero della zona 2

numero della zona 3

numero della zona 4

numero della zona 5

numero della zona 6

zona contenente un elenco di altre zone

indirezione doppia

inode come memorizzato nel file system Minix

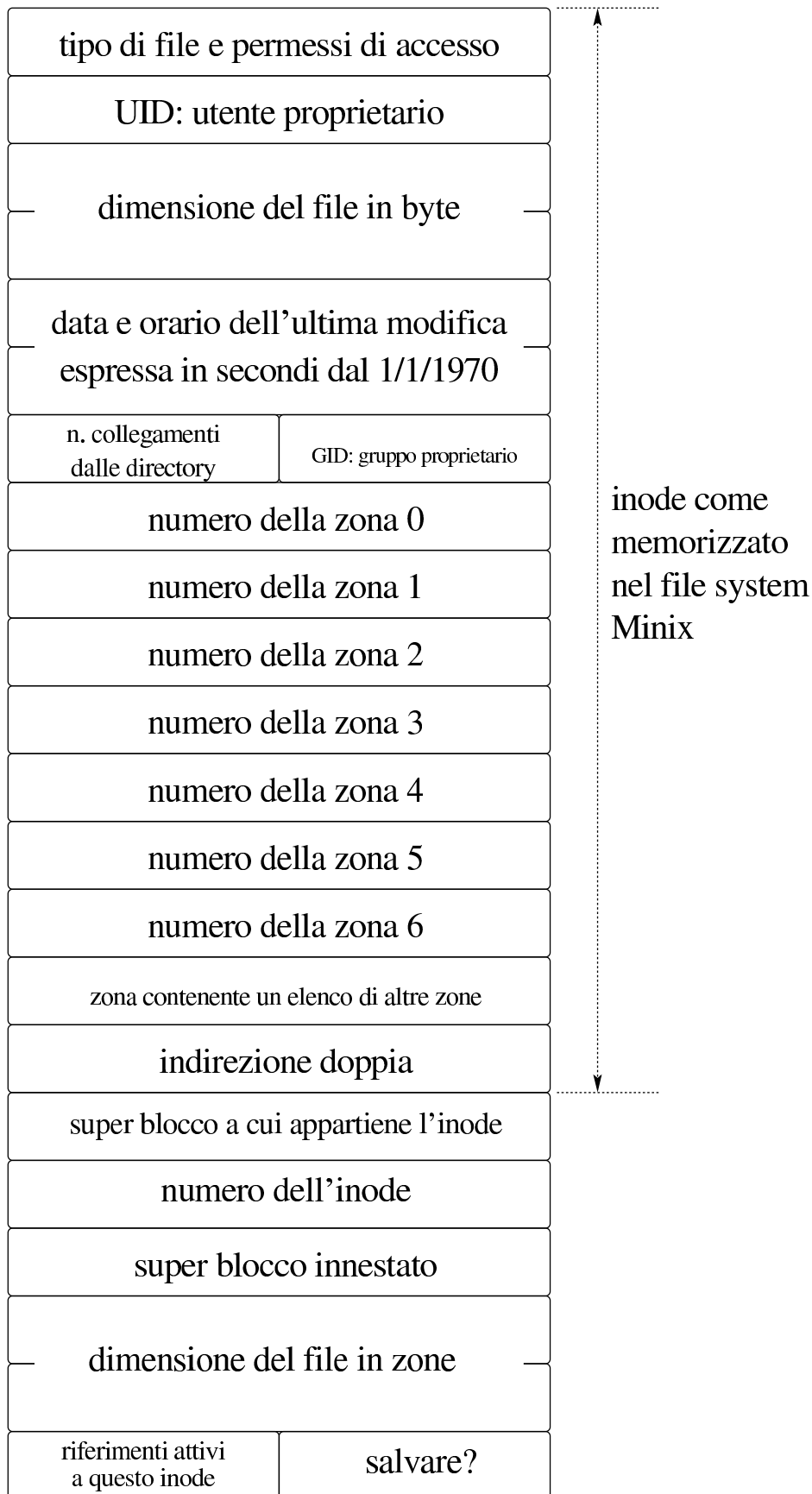
super blocco a cui appartiene l'inode

numero dell'inode

super blocco innestato

dimensione del file in zone

riferimenti attivi a questo inode

salvare?

Listato 90.48. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.

```
<verbatimpre width="60">
<![CDATA[
typedef struct inode     inode_t;

struct inode {
    mode_t         mode;
    uid_t          uid;
    ssize_t        size;
    time_t         time;
    uint8_t        gid;
    uint8_t        links;
    zno_t          direct[7];
    zno_t          indirect1;
    zno_t          indirect2;
    //------------------------
    sb_t          *sb;
    ino_t          ino;
    sb_t          *sb_attached;
    blkcnt_t       blkcnt;
    unsigned char references;
    char           changed;
};
```

Figura 90.49. Collegamento tra la tabella degli inode e quella dei super blocchi.

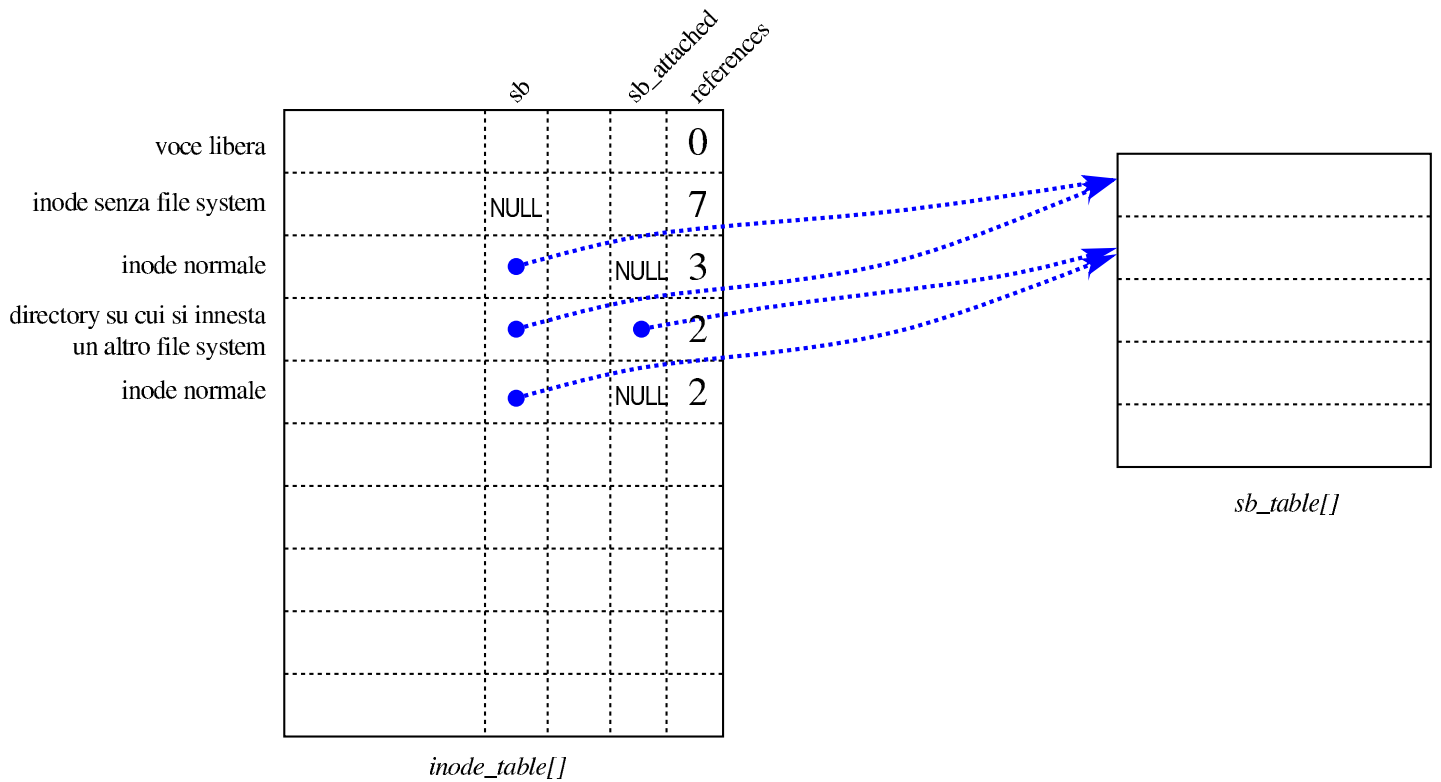



Tabella 90.50. Funzioni per la gestione dei file in forma di inode, definite nei file '`kernel/fs/inode_...`'.

Funzione	Descrizione

`inode_t *`
`inode_reference (dev_t` ***device*** `,`
`ino_t` ***ino*** `);`
	Restituisce il puntatore a un inode, rappresentato in pratica da un elemento dell'array ***inode_table[]***, corrispondente a quello con il numero di dispositivo e di inode indicati come argomenti. Se entrambi gli argomenti sono a zero, si ottiene il puntatore al primo elemento; se entrambi i valori sono pari a −1, si ottiene il puntatore al primo elemento libero; se viene indicato il dispositivo zero e l'inode numero uno, si ottiene il puntatore all'elemento corrispondente alla directory radice del file system principale.

Funzione	Descrizione

`inode_t *`
`inode_alloc (dev_t` ***device***`,`
`mode_t` ***mode***`,`
`uid_t` ***uid***`);`
	La funzione ***inode_alloc()*** cerca un inode libero nel file system del dispositivo indicato, quindi lo alloca (lo segna come utilizzato) e lo modifica aggiornando il tipo e la modalità dei permessi, oltre al proprietario del file. Se la funzione riesce nel suo intento, restituisce il puntatore all'inode in memoria, il quale rimane così aperto e disponibile per ulteriori elaborazioni.

`inode_t *`
`inode_get (dev_t` ***device***`,`
`ino_t` ***ino***`);`
	Restituisce il puntatore all'inode rappresentato dal numero di dispositivo e di inode, indicati come argomenti. Se l'inode è già presente nella tabella degli inode, la cosa si risolve nell'incremento di una unità del numero dei riferimenti di tale inode; se invece l'inode non è ancora presente, questo viene caricato dal suo file system nella tabella e gli viene attribuito inizialmente un riferimento attivo.

`int inode_put (inode_t *`***inode***`);`	Rilascia un inode che non serve più. Ciò comporta la riduzione del contatore dei riferimenti nella tabella degli inode, tenendo conto che se tale valore raggiunge lo zero, si provvede anche al suo salvataggio nel file system (ammesso che l'inode della tabella risulti modificato, rispetto alla versione presente nel file system). La funzione restituisce zero in caso di successo, oppure −1 in caso contrario.
`int inode_save (inode_t *`***inode***`);`	Salva l'inode nel file system, se questo risulta modificato.

Funzione	Descrizione
`blkcnt_t` `inode_fzones_read (inode_t *`***inode***`, zno_t `***zone_start***`, void *`***buffer***`, blkcnt_t `***blkcnt***`);`	Legge da un file, identificato attraverso il puntatore all'inode (della tabella di inode), una certa quantità di zone, a partire da una certa zona relativa al file, mettendo il risultato della lettura a partire dalla posizione di memoria rappresentata da un puntatore generico. La funzione restituisce la quantità di zone lette con successo.
`ssize_t` `inode_file_read (inode_t *`***inode***`, off_t `***offset***`, void *`***buffer***`, size_t `***count***`, int *`***eof***`);`	Legge il contenuto di un file, individuato da un inode già caricato nella tabella relativa, aggiornando eventualmente una variabile contenente l'indicatore di fine file. La funzione restituisce la quantità di byte letti con successo, oppure il valore -1 in caso di problemi.
`ssize_t` `inode_file_write (inode_t *`***inode***`, off_t `***offset***`, void *`***buffer***`, size_t `***count***`);`	Scrive una certa quantità di byte nel file individuato da un inode già caricato nella tabella relativa. La funzione restituisce la quantità di byte scritti effettivamente, oppure il valore -1 in caso di problemi.
`zno_t inode_zone (inode_t *`***inode***`, zno_t `***fzone***`, int `***write***`);`	Restituisce il numero di zona effettivo, corrispondente a un numero di zona relativo a un certo file di un certo inode. Se il parametro ***write*** è pari a zero, si intende che la zona deve esistere, quindi se questa non c'è, si ottiene semplicemente un valore pari a zero; se invece l'ultimo parametro è pari a uno, nel caso la zona cercata fosse attualmente mancante, verrebbe creata al volo nel file system.

Funzione	Descrizione
`int inode_truncate (inode_t *inode);`	Riduce la dimensione del file a cui si riferisce l'inode a zero. In pratica fa sì che le zone allocate del file siano liberate. La funzione restituisce zero se l'operazione si conclude con successo, oppure −1 in caso di problemi.
`int inode_check (inode_t *inode, mode_t type, int perm, uid_t uid);`	Verifica che l'inode sia di un certo tipo e abbia i permessi di accesso necessari a un certo utente. Nel parametro ***type*** si possono indicare più tipi validi. La funzione restituisce zero in caso di successo, ovvero di compatibilità, mentre restituisce −1 se il tipo o i permessi non sono adatti.
`int inode_dir_empty (inode_t *inode);`	Verifica se la directory a cui si riferisce l'inode è effettivamente una directory ed è vuota, nel qual caso restituisce il valore uno, altrimenti restituisce zero.

90.7.4 Fasi dell'innesto di un file system

L'innesto e il distacco di un file system, coinvolge simultaneamente la tabella dei super blocchi e quella degli inode. Si distinguono due situazioni fondamentali: l'innesto del file system principale e quello di un file system ulteriore.

Quando si tratta dell'innesto del file system principale, la tabella dei super blocchi è priva di voci e quella degli inode non contiene riferimenti a file system. La funzione ***sb_mount()*** viene chiamata indicando, come riferimento all'inode di innesto, il puntatore a una variabile puntatore contenente il valore nullo:

```
...
    inode_t *inode;
    sb_t    *sb;
    ...
    inode = NULL;
    sb = sb_mount (DEV_DSK0, &inode, MOUNT_DEFAULT);
    ...
```

La funzione ***sb_mount()*** carica il super blocco nella tabella relativa, ma trovando il riferimento all'inode di innesto nullo, provvede a caricare l'inode della directory radice dello stesso dispositivo, creando un collegamento incrociato tra le tabelle dei super blocchi e degli inode, come si vede nella figura successiva.

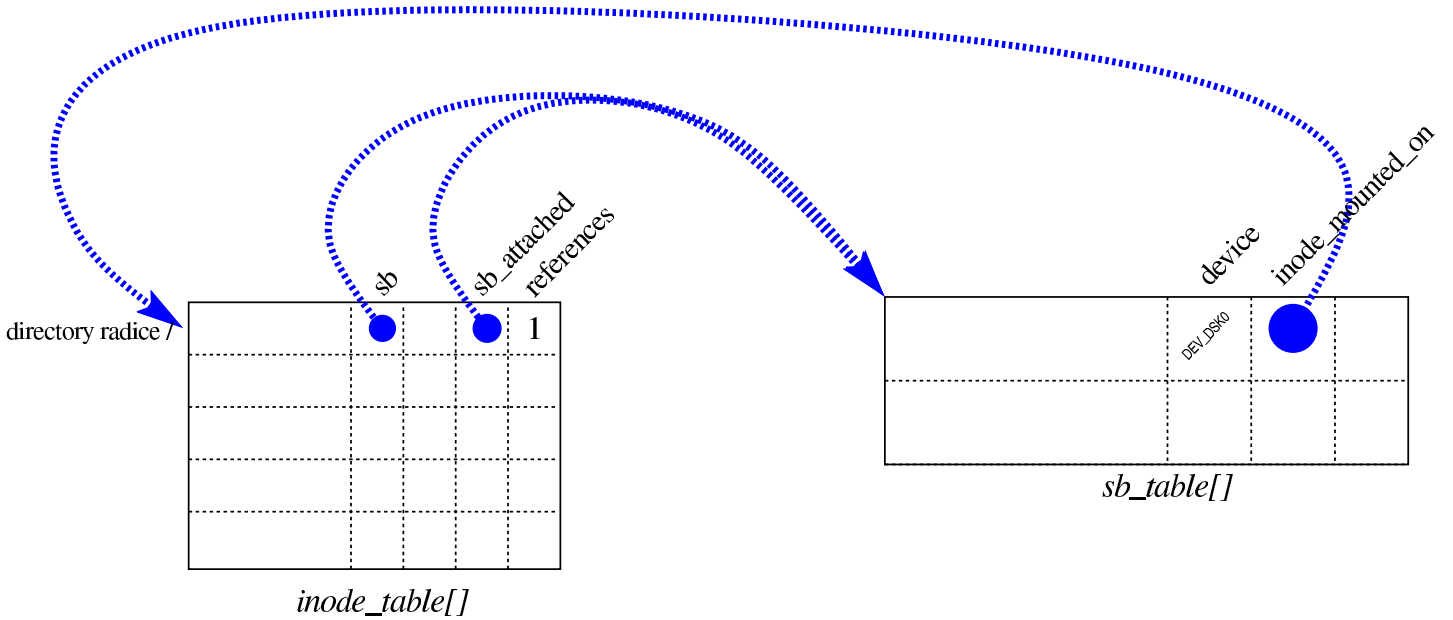
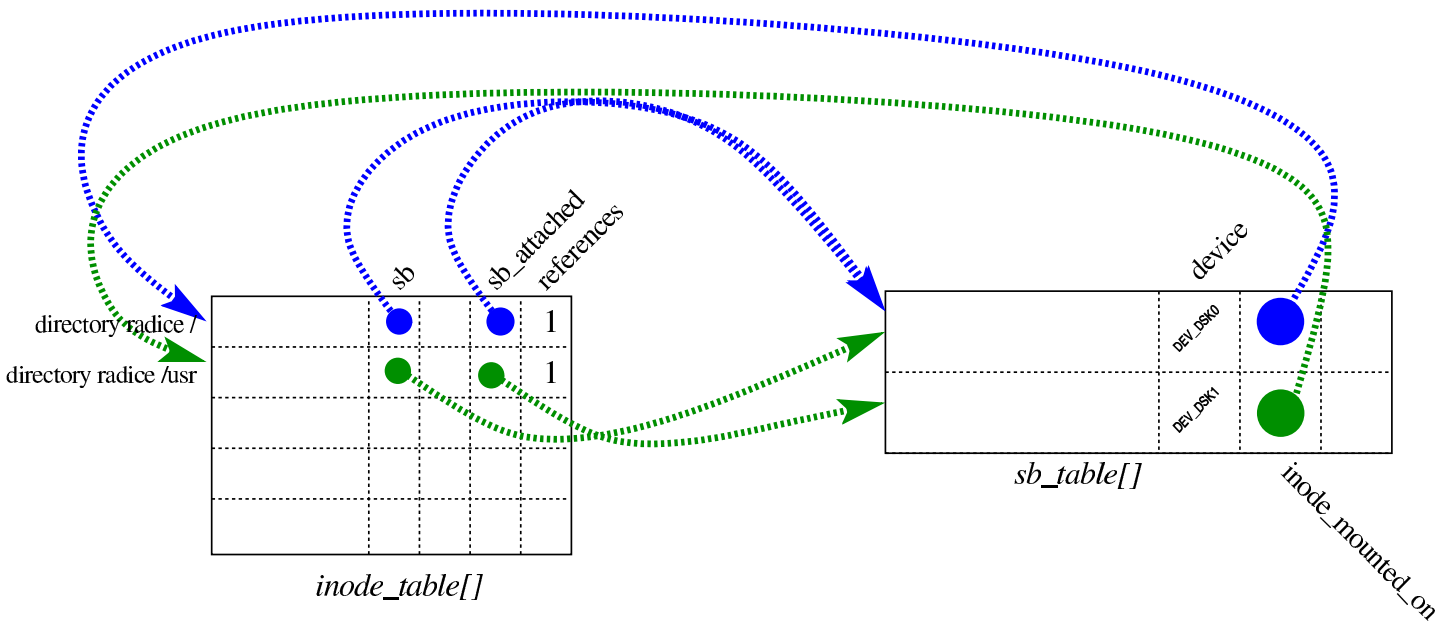
Figura 90.52. Collegamento tra la tabella degli inode e quella dei super blocchi, quando si innesta il file system principale.

Per innestare un altro file system, occorre prima disporre dell'inode di una directory (appropriata) nella tabella degli inode, quindi si può caricare il super blocco del nuovo file system, creando il collegamento tra directory e file system innestato.

Figura 90.53. Innesto di un file system nella directory '`/usr/`'.

90.7.5 File «kernel/fs/file_...»

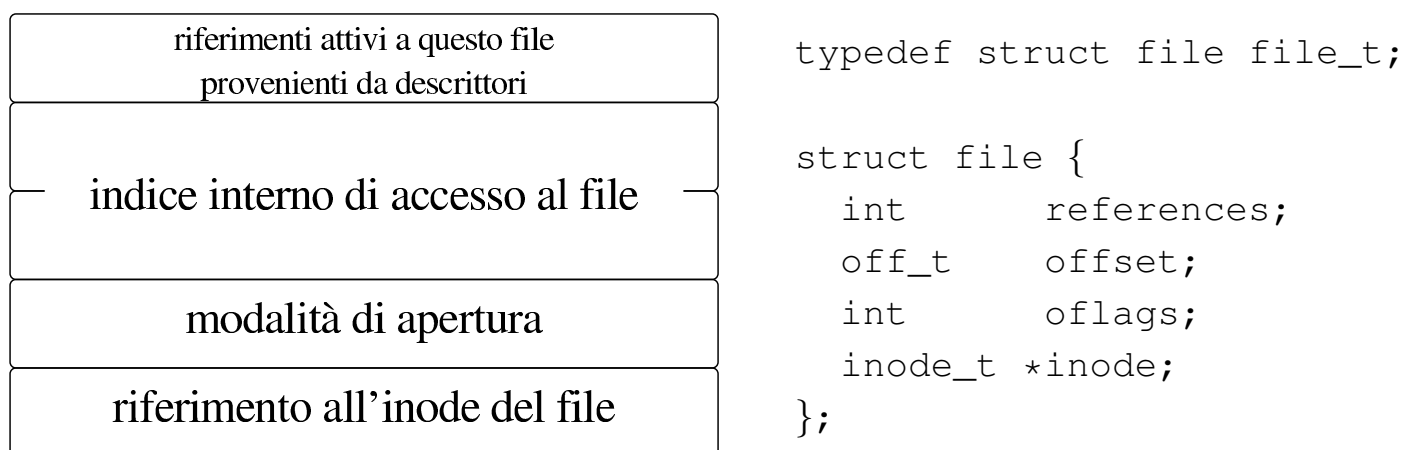
I file 'kernel/fs/file_...' descrivono le funzioni per la gestione della tabella dei file, la quale si collega a sua volta a quella degli inode. In realtà, le funzioni di questo gruppo sono in numero molto limitato, perché l'intervento nella tabella dei file avviene prevalentemente per opera di funzioni che gestiscono i descrittori.

La tabella dei file è rappresentata dall'array ***file_table[]*** e per individuare un certo elemento dell'array si usa preferibilmente la funzione ***file_reference()***. Gli elementi della tabella dei file sono di tipo '**file_t**' (definito nel file 'kernel/fs.h'); una voce della tabella rappresenta un file aperto se il campo dei riferimenti (***references***) ha un valore maggiore di zero.

Figura 90.54. Struttura del tipo '**file_t**', corrispondente agli elementi dell'array ***file_table[]***.

riferimenti attivi a questo file provenienti da descrittori
indice interno di accesso al file
modalità di apertura
riferimento all'inode del file

```
typedef struct file file_t;

struct file {
  int      references;
  off_t    offset;
  int      oflags;
  inode_t *inode;
};
```

Nel membro ***oflags*** si annotano esclusivamente opzioni relative alla modalità di apertura del file: lettura, scrittura o entrambe; pertanto si possono usare le macro-variabili ***O_RDONLY***, ***O_WRONLY*** e ***O_RDWR***, come dichiarato nel file di intestazione 'lib/fcntl.h'. Il membro ***offset*** rappresenta l'indice interno di accesso al file, per l'operazione successiva di lettura o scrittura al suo interno. Il membro ***references*** è un contatore dei riferimenti a questa tabella, da parte di descrittori di file.

La tabella dei file si collega a quella degli inode, attraverso il membro ***inode***. Più voci della tabella dei file possono riferirsi allo stesso inode, perché hanno modalità di accesso differenti, oppure soltanto per poter distinguere l'indice interno di lettura e scrittura. Va osservato che le voci della tabella di inode potrebbero essere usate direttamente e non avere elementi corrispondenti nella tabella dei file.

Figura 90.55. Collegamento tra la tabella dei file e quella degli inode.

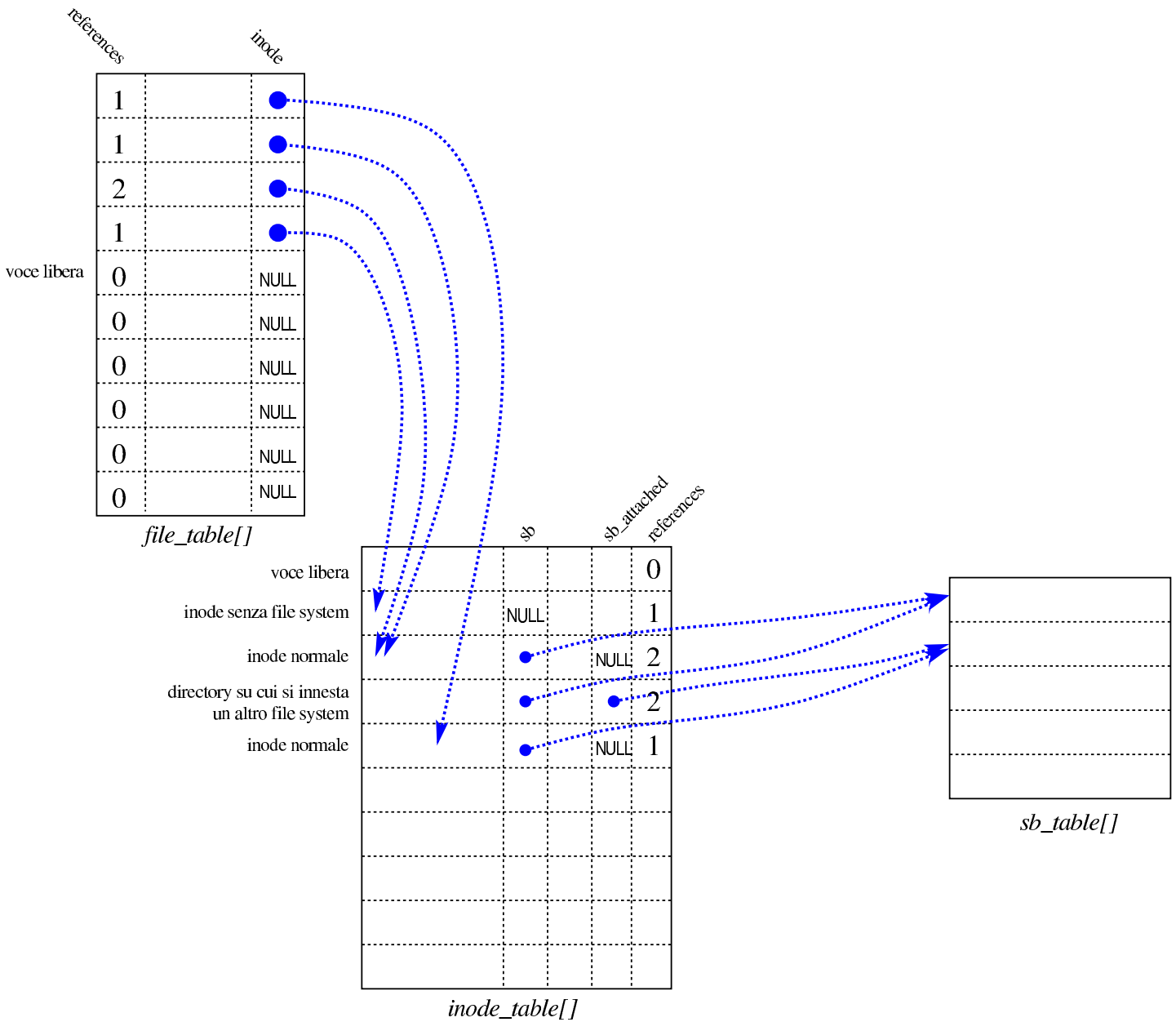



Tabella 90.56. Funzioni fatte esclusivamente per la gestione della tabella dei file ***file_table[]***.

Funzione	Descrizione
`file_t *` `file_reference (int` ***fno***`);`	Restituisce il puntatore all'elemento ***fno***-esimo della tabella dei file. Se ***fno*** è un valore negativo, viene restituito il puntatore a una voce libera della tabella.

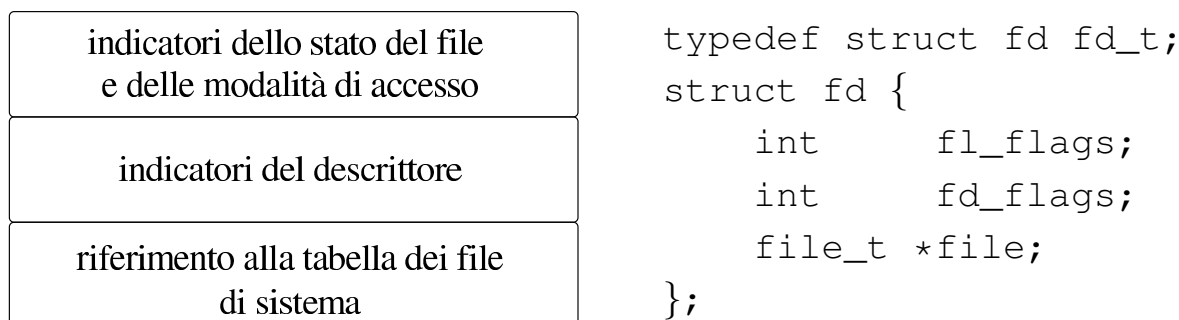
`file_t *`
`file_stdio_dev_make (dev_t` ***device***`,`
`mode_t` ***mode***`,`
`int` ***oflags***`);`
	Crea una voce per l'accesso a un file di dispositivo standard di input-output, restituendo il puntatore alla voce stessa.

90.7.6 Descrittori di file

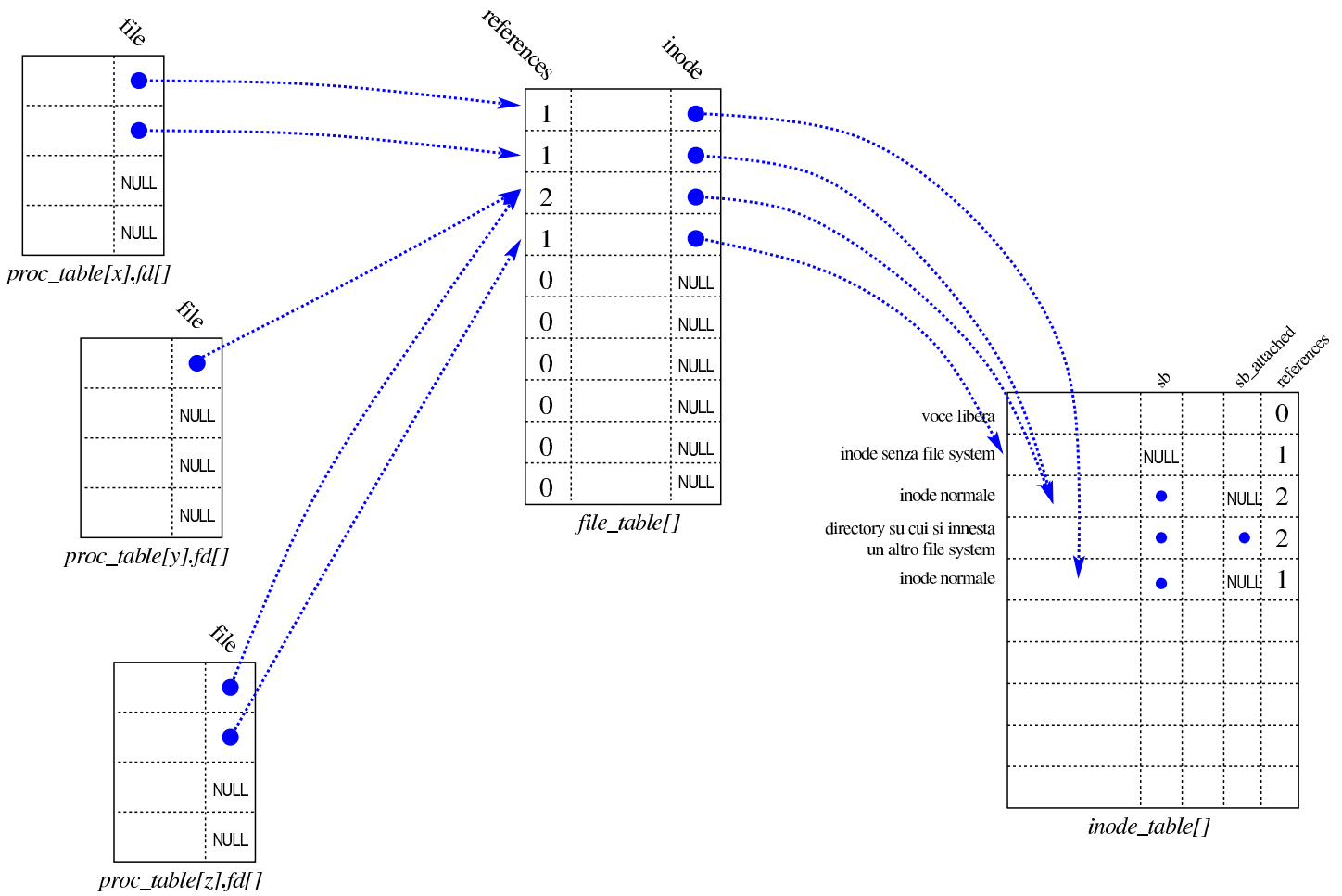
Le tabelle di super blocchi, inode e file, riguardano il sistema nel complesso. Tuttavia, l'accesso normale ai file avviene attraverso il concetto di «descrittore», il quale è un file aperto da un certo processo elaborativo. Nel file 'kernel/fs.h' si trova la dichiarazione e descrizione del tipo derivato '**fd_t**', usato per costruire una tabella di descrittori, ma tale tabella non fa parte della gestione del file system, bensì è incorporata nella tabella dei processi elaborativi. Pertanto, ogni processo ha una propria tabella di descrittori di file.

Figura 90.57. Struttura del tipo '**fd_t**', con cui si costituiscono gli elementi delle tabelle dei descrittori di file, una per ogni processo.

indicatori dello stato del file e delle modalità di accesso
indicatori del descrittore
riferimento alla tabella dei file di sistema

```
typedef struct fd fd_t;
struct fd {
    int      fl_flags;
    int      fd_flags;
    file_t *file;
};
```

Il membro ***fl_flags*** consente di annotare indicatori del tipo '**O_RDONLY**', '**O_WRONLY**', '**O_RDWR**', '**O_CREAT**', '**O_EXCL**', '**O_NOCTTY**', '**O_TRUNC**' e '**O_APPEND**', come dichiarato nella libreria standard, nel file di intestazione 'lib/fcntl.h'. Tali indicatori si combinano assieme con l'operatore binario OR. Altri tipi di opzione che sarebbero previsti nel file 'lib/fcntl.h', sono privi di effetto nella gestione del file system di os16.

Il membro ***fd_flags*** serve a contenere, eventualmente, l'opzione '**FD_CLOEXEC**', definita nel file 'lib/fcntl.h'. Non sono previste altre opzioni di questo tipo.

Figura 90.58. Collegamento tra le tabelle dei descrittori e la tabella complessiva dei file. La tabella ***proc_table[x].fd[]*** rappresenta i descrittori di file del processo elaborativo ***x***.

90.7.7 File «kernel/fs/path_...»

I file '`kernel/fs/path_...`' descrivono le funzioni che fanno riferimento a file o directory attraverso una stringa che ne descrive il percorso.

Tabella 90.59. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, senza indicazioni sul processo elaborativo.

Funzione	Descrizione
`int path_fix (char *`***path***`);`	Verifica il percorso indicato semplificandolo, quindi sovrascrive il percorso originario con quello riveduto e corretto. Un percorso assoluto rimane assoluto; un percorso relativo rimane relativo, mancando qualunque indicazione sulla directory corrente.

Funzione	Descrizione

`int path_full (const char *`***path***`,`
`const char *`***path_cwd***`,`
`char *`***full_path***`);`
	Ricostruisce un percorso assoluto, usando come riferimento la directory corrente indicata in ***path_cwd***, salvandolo in ***path_full***.

Tabella 90.60. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, tenendo conto del processo elaborativo per conto del quale si svolge l'operazione. Del processo elaborativo si considera soprattutto l'identità efficace, per conoscerne i privilegi e determinare se è data effettivamente la facoltà di eseguire l'azione richiesta.

Funzione	Descrizione
`int path_chdir (pid_t `***pid***`,` `const char *`***path***`);`	Cambia la directory corrente, utilizzando il nuovo percorso indicato. È l'equivalente della funzione standard ***chdir()*** (sezione 93.3).

`int path_chmod (pid_t `***pid***`,`
`const char *`***path***`,`
`mode_t `***mode***`);`
	Cambia la modalità di accesso al file indicato. È l'equivalente della funzione standard ***chmod()*** (sezione 93.4).

`int path_chown (pid_t `***pid***`,`
`const char *`***path***`,`
`uid_t `***uid***`,`
`gid_t `***gid***`);`
	Cambia l'utente e il gruppo proprietari del file (va però ricordato che os16 non considera i gruppi, anche se nel file system sono annotati). È l'equivalente della funzione standard ***chown()*** (sezione 93.5).

`dev_t path_device (pid_t `***pid***`,` `const char *`***path***`);`	Restituisce il numero del dispositivo di un file di dispositivo; pertanto, il percorso deve fare riferimento a un file di dispositivo, per poter ottenere un risultato valido.

Funzione	Descrizione

`inode_t *`
`path_inode (pid_t` ***pid***`,`
`const char *`***path***`);`
	Apre l'inode del file indicato tramite il percorso, purché il processo ***pid*** abbia i permessi di accesso («x») alle directory che vi conducono. La funzione restituisce il puntatore all'inode aperto, oppure il puntatore nullo se non può eseguire l'operazione.

`inode_t`
`*path_inode_link (pid_t` ***pid***`,`
`const char *`***path***`,`
`inode_t *`***inode***`,`
`mode_t` ***mode***`);`
	Crea un collegamento fisico con il nome fornito in ***path***, riferito all'inode a cui punta ***inode***, ma se ***inode*** fosse un puntatore nullo, verrebbe semplicemente creato un file vuoto con un nuovo inode. Si richiede inoltre che il processo ***pid*** abbia i permessi di accesso per tutte le directory che portano al file da collegare e che nell'ultima ci sia anche il permesso di scrittura, dovendo intervenire su tale directory in questo modo. Se la funzione riesce nel proprio intento, restituisce il puntatore a ciò che descrive l'inode collegato o creato.

`int`
`path_link (pid_t` ***pid***`,`
`const char *`***path_old***`,`
`const char *`***path_new***`);`
	Crea un collegamento fisico. È l'equivalente della funzione standard ***link()*** (sezione 93.23).

`int path_mkdir (pid_t` ***pid***`,`
`const char *`***path***`,`
`mode_t` ***mode***`);`
	Crea una directory, con la modalità dei permessi indicata. È l'equivalente della funzione standard ***mkdir()*** (sezione 93.25).

Funzione	Descrizione

`int path_mknod (pid_t` ***pid***`,`
`const char *`***path***`,`
`mode_t` ***mode***`,`
`dev_t` ***device***`);`
	Crea un file vuoto, con il tipo e i permessi specificati da ***mode***; se si tratta di un file di dispositivo, viene preso in considerazione anche il parametro ***device***, per specificare il numero primario e secondario dello stesso. Va osservato che con questa funzione è possibile creare una directory priva delle voci '`.`' e '`..`'. È l'equivalente della funzione standard ***mknod()*** (sezione 93.26).

`int path_stat (pid_t` ***pid***`,`
`const char *`***path***`,`
`struct stat *`***buffer***`);`
	Aggiorna la variabile strutturata a cui punta ***buffer***, con le informazioni sul file specificato. È l'equivalente della funzione standard ***stat()*** (sezione 93.36).

`int path_unlink (pid_t` ***pid***`,` `const char *`***path***`);`	Cancella un file o una directory, purché questa sia vuota. È l'equivalente della funzione standard ***unlink()*** (sezione 93.42).

`int`
`path_mount (pid_t` ***pid***`,`
`const char *`***path_dev***`,`
`const char *`***path_mnt***`,`
`int` ***options***`);`
	Innesta il dispositivo corrispondente a ***path_dev***, nella directory ***path_mnt*** (tenendo conto della directory corrente del processo ***pid***), con le opzioni specificate, per conto dell'utente ***uid***. Le opzioni disponibili sono solo '**`MOUNT_DEFAULT`**' e '**`MOUNT_RO`**', come dichiarato nel file di intestazione '`lib/sys/os16.h`'.

`int`
`path_umount (pid_t` ***pid***`,`
`const char *`***path_mnt***`);`
	Stacca l'unità innestata nella directory indicata, purché nulla al suo interno sia attualmente in uso.

90.7.8 File «kernel/fs/fd_...»

«

I file '`kernel/fs/fd_...`' descrivono le funzioni che fanno riferimento a file o directory attraverso il numero di descrittore, riferito a sua volta a un certo processo elaborativo. Pertanto, il numero del processo e il numero del descrittore sono i primi due parametri obbligatori di tutte queste funzioni.

Tabella 90.61. Funzioni per la gestione dei file, a cui si fa riferimento attraverso il descrittore, relativamente a un certo processo elaborativo. La funzione ***fd_open()*** fa eccezione, in quanto apre un descrittore, ma per identificare il file non ancora aperto, ne richiede il percorso.

Funzione	Descrizione
`int fd_chmod (pid_t` ***pid***`,` `int` ***fdn***`,` `mode_t` ***mode***`);`	Cambia la modalità dei permessi (solo gli ultimi 12 bit del parametro ***mode*** vengono considerati). È l'equivalente della funzione standard ***fchmod()*** (sezione 93.4).
`int fd_chown (pid_t` ***pid***`,` `int` ***fdn***`,` `uid_t` ***uid***`,` `gid_t` ***gid***`);`	Cambia la proprietà (utente e gruppo). È l'equivalente della funzione standard ***fchown()*** (sezione 93.5).
`int fd_close (pid_t` ***pid***`,` `int` ***fdn***`);`	Chiude il descrittore di file. È l'equivalente della funzione standard ***close()*** (sezione 93.7).
`int fd_dup (pid_t` ***pid***`,` `int` ***fdn_old***`,` `int` ***fdn_min***`);`	Duplica il descrittore ***fdn_old***, creandone un altro con numero maggiore o uguale a ***fdn_min*** (viene scelto il primo libero a partire da ***fdn_num***). È l'equivalente della funzione standard ***dup()*** (sezione 93.8).
`int fd_dup2 (pid_t` ***pid***`,` `int` ***fdn_old***`,` `int` ***fdn_new***`);`	Duplica il descrittore ***fdn_old***, creandone un altro con numero ***fdn_new***. Se però ***fdn_new*** è già aperto, prima della duplicazione questo viene chiuso. È l'equivalente della funzione standard ***dup2()*** (sezione 93.8).

Funzione	Descrizione

`int fd_fcntl (pid_t` ***pid***`,`
`int` ***fdn***`,`
`int` ***cmd***`,`
`int` ***arg***`);`
	Svolge il compito della funzione standard ***fcntl()*** (sezione 93.13).

`off_t fd_lseek (pid_t` ***pid***`,`
`int` ***fdn***`,`
`off_t` ***offset***`,`
`int` ***whence***`);`
	Riposiziona l'indice interno di accesso del descrittore di file. È l'equivalente della funzione standard ***lseek()*** (sezione 93.24).

`int fd_open (pid_t` ***pid***`,`
`const char *`***path***`,`
`int` ***oflags***`,`
`mode_t` ***mode***`);`
	Apre un descrittore, fornendo però il percorso del file. È l'equivalente della funzione standard ***open()*** (sezione 93.28).

`ssize_t fd_read (pid_t` ***pid***`,`
`int` ***fdn***`,`
`void *`***buffer***`,`
`size_t` ***count***`,`
`int *`***eof***`);`
	Legge da un descrittore, aggiornando eventualmente la variabile ****eof*** in caso di fine del file. È l'equivalente della funzione standard ***read()*** (sezione 93.29).

`fd_t *fd_reference (pid_t` ***pid***`,` `int *`***fdn***`);`	Produce il puntatore ai dati del descrittore ****fdn***. Se ****fdn*** è minore di zero, si ottiene il riferimento al primo descrittore libero, aggiornando anche ****fdn*** stesso.

`int fd_stat (pid_t` ***pid***`,`
`int` ***fdn***`,`
`struct stat *`***buffer***`);`
	Svolge il compito della funzione standard ***fstat()*** (sezione 93.36).

`ssize_t`
`fd_write (pid_t` ***pid***`,`
`int` ***fdn***`,`
`const void *`***buffer***`,`
`size_t` ***count***`);`
	Scrive nel descrittore. È l'equivalente della funzione standard ***write()*** (sezione 93.44).

90.8 Gestione dei processi

La gestione dei processi è raccolta nei file '`kernel/proc.h`' e '`kernel/proc/...`', dove il file '`kernel/proc/_isr.s`', in particolare, contiene il codice attivato dalle interruzioni. Nella semplicità di os16, ci sono solo due interruzioni che vengono gestite: quella del temporizzatore il quale produce un impulso 18,2 volte al secondo, e quella causata dalle chiamate di sistema.

Con os16, quando un processo viene interrotto, per lo svolgimento del compito dell'interruzione, si passa sempre a utilizzare la pila dei dati del kernel. Per annotare la posizione in cui si trova l'indice della pila del kernel si usa la variabile ***_ksp***, accessibile anche dal codice in linguaggio C.

Il codice del kernel può essere interrotto dagli impulsi del temporizzatore, ma in tal caso non viene coinvolto lo schedulatore per lo scambio con un altro processo, così che dopo l'interruzione è sempre il kernel che continua a funzionare; pertanto, nella funzione ***main()*** è il kernel che cede volontariamente il controllo a un altro processo (ammesso che ci sia) con una chiamata di sistema nulla.

90.8.1 File «kernel/proc/_isr.s» e «kernel/proc/_ivt_load.s»

Listati 100.9.1 e 100.9.2.

Il file '`kernel/proc/_isr.s`' contiene il codice per la gestione delle interruzioni dei processi. Nella parte iniziale del file, vengono dichiarate delle variabili, alcune delle quali sono pubbliche e accessibili anche dal codice in C.

```
...
proc_ss_0:          .word 0x0000
proc_sp_0:          .word 0x0000
proc_ss_1:          .word 0x0000
proc_sp_1:          .word 0x0000
proc_syscallnr:     .word 0x0000
proc_msg_offset:    .word 0x0000
proc_msg_size:      .word 0x0000
__ksp:              .word 0x0000
__clock_ticks:
ticks_lo:           .word 0x0000
ticks_hi:           .word 0x0000
__clock_seconds:
seconds_lo:         .word 0x0000
seconds_hi:         .word 0x0000
...
```

Si tratta di variabili scalari da 16 bit, tenendo conto che: i simboli '**`ticks_lo`**' e '**`ticks_hi`**' compongono assieme la variabile ***_clock_ticks*** a 32 bit per il linguaggio C; i simboli '**`seconds_lo`**' e '**`seconds_hi`**' compongono assieme la variabile ***_clock_seconds*** a 32 bit per il linguaggio C.

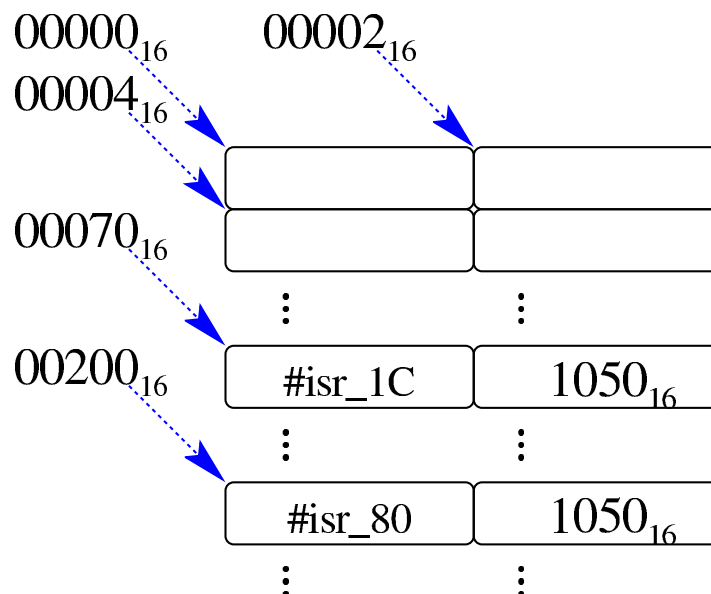
Dopo la dichiarazione delle variabili inizia il codice vero e proprio. Il simbolo '**`isr_1C`**' si riferisce al codice da usare in presenza dell'interruzione $1C_{16}$, mentre il simbolo '**`isr_80`**' riguarda l'interruzione 80_{16}.

Nel file '`kernel/proc/_ivt_load.s`', la funzione ***_ivt_load()*** che inizia con il simbolo '**`__ivt_load`**', modifica la tabella IVT (*Interrupt vector table*) in modo che le interruzioni $1C_{16}$ e 80_{16} portino all'esecuzione del codice che inizia rispettivamente in corrispondenza dei simboli '**`isr_1C`**' e '**`isr_80`**' (del file '`kernel/proc/_isr.s`').

```
...
__ivt_load:
    enter #0, #0          ; No local variables.
    pushf
    cli
    pusha
    ;
    mov   ax,   #0        ; Change the DS segment to 0.
    mov   ds,   ax        ;
    ;
    mov   bx,   #112      ; Timer      INT 0x08 (8) --> 0x1C
    mov   [bx], #isr_1C   ; offset
    mov   bx,   #114      ;
    mov   [bx], cs        ; segment
    ;
    mov   bx,   #512      ; Syscall    INT 0x80 (128)
    mov   [bx], #isr_80   ; offset
    mov   bx,   #514      ;
    mov   [bx], cs        ; segment
    ;
    mov   ax,   #0x0050   ; Put the DS segment back to the
    mov   ds,   ax        ; right value.
    ;
    popa
    popf
    leave
```

```
ret
```

Per compiere il suo lavoro, la funzione ***_ivt_load()*** salva inizialmente lo stato degli indicatori contenuti nel registro ***FLAGS*** e gli altri registri principali, quindi modifica il registro ***DS*** in modo che il segmento dati corrisponda allo zero, per poter accedere al contenuto della tabella IVT (che inizia proprio dall'indirizzo 00000_{16}). A quel punto, all'indirizzo efficace 00070_{16} (112_{10}) scrive l'indirizzo relativo del simbolo '**`isr_1C`**' (l'indirizzo relativo al segmento codice attuale) e il valore del segmento codice all'indirizzo efficace 00072_{16} (114_{10}). Nello stesso modo agisce per il simbolo '**`isr_80`**', scrivendo il suo indirizzo relativo all'indirizzo efficace 00200_{16} (512_{10}), assieme al valore del segmento codice che va invece in 00202_{16} (514_{10}). In tal modo, quando scatta l'interruzione $1C_{16}$ che deriva dalla scansione del temporizzatore interno, viene eseguito il codice che si trova nella voce corrispondente della tabella IVT, ovvero, proprio ciò che comincia con il simbolo '**`isr_1C`**', mentre quando scatta l'interruzione 80_{16} si ottiene l'esecuzione del codice che si trova a partire dal simbolo '**`isr_80`**'.

Figura 90.64. Modifica della tabella IVT attraverso la funzione ***_ivt_load()***. Il valore del segmento codice è sicuramente 1050_{16}, in quanto si tratta di quello del kernel, il quale va a collocarsi in quella posizione.

Le interruzioni previste con os16 sono solo due: quella del temporizzatore (*timer*) che invia un impulso a 18,2 Hz circa e quella che serve per le chiamate di sistema. Per la precisione, il temporizzatore fa scattare l'interruzione 08_{16}, ma se si utilizza il codice del BIOS, non può essere ridiretta; pertanto, il codice predefinito per tale interruzione, al termine del suo compito, fa scattare l'interruzione $1C_{16}$, la quale può essere ridiretta come appena mostrato.

Il codice per le due interruzioni gestite è simile, con la differenza fondamentale che

per l'interruzione proveniente dal temporizzatore si incrementano i contatori rappresentati dalle variabili ***_clock_ticks*** e ***_clock_seconds***. Il codice equivalente della gestione delle due interruzioni è il seguente:

```
…
isr_1C:|isr_80:
    push   es    ; extra segment
    push   ds    ; data segment
    push   di    ; destination index
    push   si    ; source index
    push   bp    ; base pointer
    push   bx    ; BX
    push   dx    ; DX
    push   cx    ; CX
    push   ax    ; AX
    ;
    mov ax, #0x0050  ; DS and ES.
    mov ds, ax       ;
    mov es, ax       ;
    …
    …
    pop    ax
    pop    cx
    pop    dx
    pop    bx
    pop    bp
    pop    si
    pop    di
    pop    ds
    pop    es
    ;
    iret
…
```

Mentre viene eseguito il codice che si trova a partire da '**`isr_1C`**' o da '**`isr_80`**', il segmento codice è quello del kernel, ma quello dei dati è quello del processo che è stato interrotto poco prima. Nella pila dei dati di quel processo, nel momento in cui viene raggiunto questo codice ci sono già i valori di alcuni registri, nello stato in cui erano al verificarsi dell'interruzione: ***FLAGS***, ***CS***, ***IP***. Come si vede dal codice

appena mostrato, si aggiungono nella pila altri registri.

Figura 90.66. Inserimento nella pila del processo interrotto.

inizio della routine

AX
CX
DX
BX
BP
SI
DII
DS
ES
IP
CS
FLAGS

inserimento all'inizio della routine ISR

inserimento dovuto all'interruzione

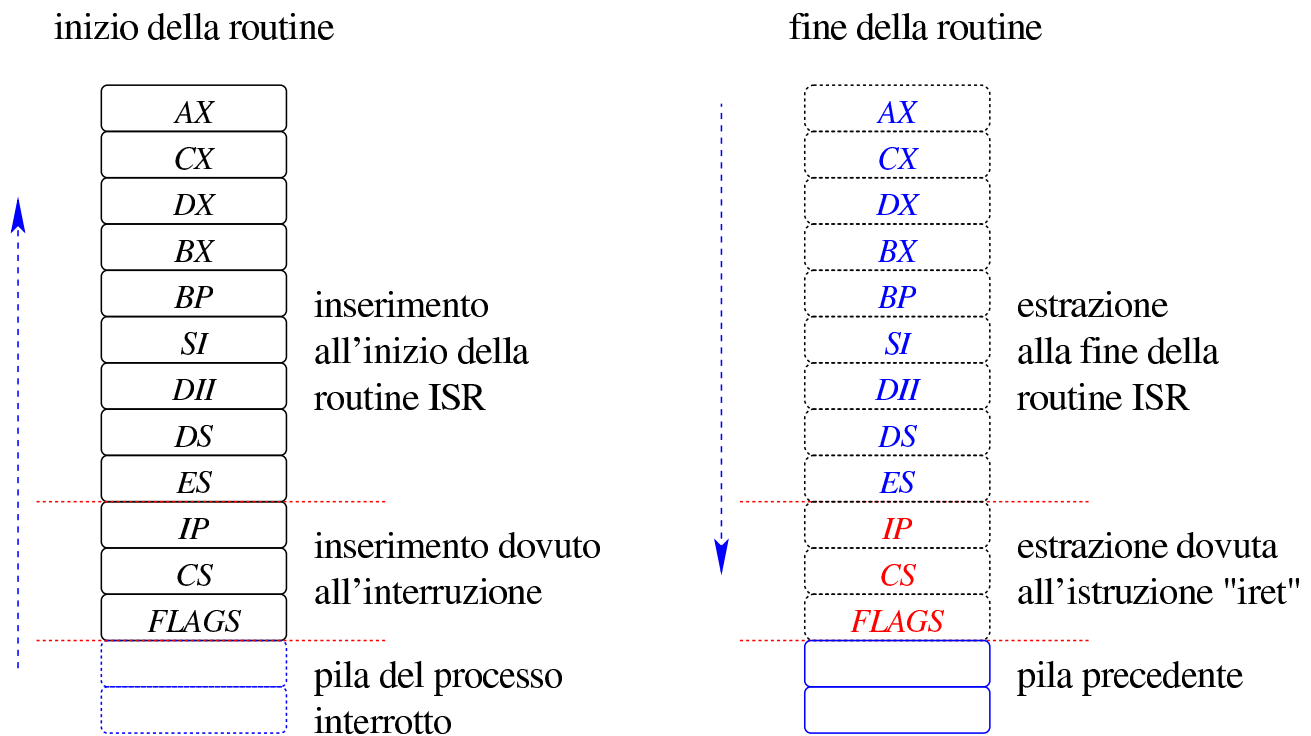
pila del processo interrotto

fine della routine

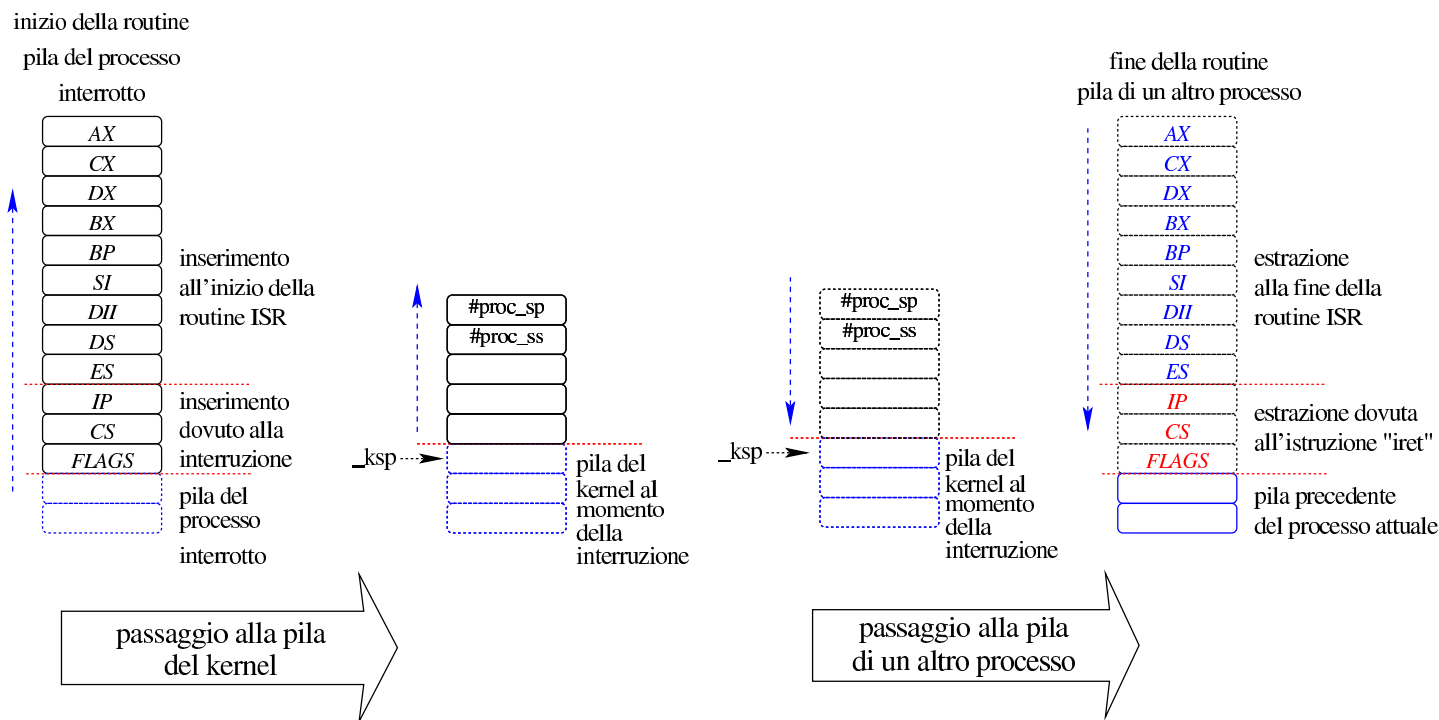
AX
CX
DX
BX
BP
SI
DII
DS
ES
IP
CS
FLAGS

estrazione alla fine della routine ISR

estrazione dovuta all'istruzione "iret"

pila precedente

Dopo il salvataggio nella pila dei registri principali, viene modificato il valore dei registri ***DS*** e ***ES***, per consentire l'accesso alle variabili dichiarate all'inizio del file '`kernel/_isr.s`'. Il valore che si attribuisce a tali registri è 0050_{16}, perché il segmento dati del kernel inizia all'indirizzo efficace 00500_{16}. Va osservato che il segmento usato per la pila dei dati non viene ancora modificato e rimane nel segmento dati del processo interrotto.

A questo punto iniziano le differenze tra le due routine di gestione delle interruzioni. In ogni caso rimane il principio di massima, descritto intuitivamente dalla figura successiva, per cui si scambia la pila del processo interrotto con quella del kernel, poi si esegue la chiamata di sistema o si attiva lo schedulatore, quindi si passa nuovamente alla pila di un processo, il quale può essere diverso da quello interrotto.

Figura 90.67. Scambi delle pile.

90.8.1.1 Routine «isr_1C»

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, la routine '**isr_1C**' si occupa di incrementare i contatori degli impulsi e dei secondi:

```
...
isr_1C:
    ...
    add ticks_lo, #1      ; Clock ticks counter.
    adc ticks_hi, #0      ;
    ;
    mov dx, ticks_hi      ;
    mov ax, ticks_lo      ; DX := ticks % 18
    mov cx, #18           ;
    div cx                ;
    mov ax, #0            ; If the ticks value can be divided
    cmp ax, dx            ; by 18, the seconds is incremented
    jnz L1                ; by 1.
    add seconds_lo, #1    ;
    adc seconds_hi, #0    ;
    ;
L1:
    ...
```

Per semplificare i calcoli, si considera che ogni 18 impulsi sia trascorso un secondo e di conseguenza va interpretata la divisione che viene eseguita. In ogni caso, quando si arriva al simbolo '**L1**' le variabili sono state aggiornate correttamente.

A questo punto viene salvato il valore del segmento in cui si trova la pila dei dati e l'indice all'interno della stessa, usando delle variabili locali, le quali non sono però accessibili dal codice in linguaggio C:

```
...
L1:
    mov proc_ss_0, ss   ; Save process stack segment.
    mov proc_sp_0, sp   ; Save process stack pointer.
    ...
```

Poi si verifica se la pila dei dati del processo interrotto si trova nel kernel. In tal caso, il suo segmento avrebbe il valore 0050_{16}. Se il segmento dati è proprio quello del kernel, si saltano le istruzioni successive, riprendendo dal ripristino dei registri dalla pila dei dati (dal simbolo '**L2**').

```
    ...
    mov dx, proc_ss_0
    mov ax, #0x0050     ; Kernel data area.
    cmp dx, ax
    je L2
    ...
```

Se non è il kernel che è stato interrotto, si fa in modo di saltare all'utilizzo della pila dei dati del kernel. Per fare questo viene sostituito il valore del registro '**SS**', facendo in modo che corrisponda al segmento dati del kernel stesso, quindi si modifica il valore del registro '**SP**', mettendovi il valore salvato precedentemente nella variabile ***_ksp*** (ovvero il simbolo '**__ksp**').

```
    ...
    mov ax, #0x0050     ; Kernel data area.
    mov ss, ax
    mov sp, __ksp
    ...
```

Nella variabile ***_ksp*** c'è sicuramente l'indice della pila del kernel, aggiornata dalla funzione ***proc_scheduler()***. Tale aggiornamento della variabile ***_ksp*** avviene quando

il gestore dei processi elaborativi sospende il codice del kernel per mettere in funzione un altro processo.

A questo punto, il contesto esecutivo è diventato quello del kernel, provenendo però dall'interruzione di un altro processo. Quindi viene chiamata la funzione di attivazione dello schedulatore: ***proc_scheduler()***. Tale funzione richiede dei parametri e gli vengono forniti i puntatori alle variabili contenenti il segmento e l'indice della pila dei dati del processo interrotto.

```
...
push #proc_ss_0     ; &proc_ss_0
push #proc_sp_0     ; &proc_sp_0
call _proc_scheduler
add  sp, #2
add  sp, #2
...
```

Al termine del lavoro della funzione ***proc_scheduler()***, i valori contenuti nelle variabili rappresentate dai simboli '**`proc_ss_0`**' e '**`proc_sp_0`**' possono essere stati sostituiti con quelli di un altro processo da attivare al posto di quello interrotto precedentemente. Infatti, i registri ***SS*** e ***SP*** vengono sostituiti subito dopo:

```
...
mov ss, proc_ss_0   ; Restore process stack segment.
mov sp, proc_sp_0   ; Restore process stack pointer.
...
```

Infine, si ripristinano gli altri registri, traendo i dati dalla nuova pila.

90.8.1.2 Routine «isr_80»

«

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, la routine '**`isr_80`**' salva il valore del segmento in cui si trova la pila dei dati e l'indice all'interno della stessa, usando delle variabili locali, le quali non sono però accessibili dal codice in C:

```
...
mov proc_ss_1, ss   ; Save process stack segment.
mov proc_sp_1, sp   ; Save process stack pointer.
...
```

Vengono quindi salvati dei dati contenuti ancora nella pila attuale, utilizzando delle variabili statiche, che però non sono accessibili dal codice C:

```
    …
    mov bp, sp
    mov ax, +26[bp]
    mov proc_syscallnr,  ax
    mov ax, +28[bp]
    mov proc_msg_offset, ax
    mov ax, +30[bp]
    mov proc_msg_size,   ax
    …
```

Finalmente si passa a verificare se il processo interrotto è il kernel o meno: se si tratta proprio del kernel, il valore del registro ***SP*** viene salvato nella variabile ***_ksp***.

```
    …
    mov dx, ss
    mov ax, #0x0050      ; Kernel data area.
    cmp dx, ax
    jne L3
    mov __ksp, sp
L3:
    …
```

Successivamente si scambia la pila dei dati attuale, passando a quella del kernel, utilizzando la variabile ***_ksp*** per modificare il registro ***SP***. Naturalmente si comprende che se il codice interrotto era già quello del kernel, la sostituzione non cambia in pratica i valori che già avevano i registri ***SS*** e ***SP***:

```
    …
L3:
    mov ax, #0x0050      ; Kernel data area.
    mov ss, ax
    mov sp, __ksp
    …
```

Quando la pila dei dati in funzione è quella del kernel, si passa alla chiamata della funzione ***sysroutine()***, passandole come parametri i dati raccolti precedentemente dalla pila del processo interrotto, fornendo anche i puntatori alle variabili che contengono

i dati necessari a raggiungere tale pila.

```
…
push proc_msg_size
push proc_msg_offset
push proc_syscallnr
push #proc_ss_1      ; &proc_ss_1
push #proc_sp_1      ; &proc_sp_1
call _sysroutine
add  sp, #2
add  sp, #2
add  sp, #2
add  sp, #2
add  sp, #2
…
```

La funzione ***sysroutine()*** chiama a sua volta la funzione ***proc_scheduler()***, la quale può modificare il contenuto delle variabili rappresentate dai simboli '**proc_ss_1**' e '**proc_sp_1**'; pertanto, quando i valori di tali variabili vengono usati per rimpiazzare il contenuto dei registri ***SS*** e ***SP***, si ottiene lo scambio a un processo diverso da quello interrotto inizialmente.

```
…
mov ss, proc_ss_1    ; Restore process stack segment.
mov sp, proc_sp_1    ; Restore process stack pointer.
…
```

Infine, si ripristinano gli altri registri, traendo i dati dalla nuova pila.

90.8.2 La tabella dei processi

«

Listato 100.9.

Nel file 'kernel/proc.h' viene definito il tipo '**proc_t**', con il quale, nel file 'kernel/proc/proc_table.c' si definisce la tabella dei processi, rappresentata dall'array ***proc_table[]***.

Figura 90.80. Struttura del tipo '**`proc_t`**', corrispondente agli elementi dell'array ***proc_table[]***.

ppid: pid genitore
pgrp: pid del gruppo
uid: identità reale
euid: identità efficace
sid: identitàsalvata
terminale di controllo
cwd: directory corrente in forma di percorso
cwd: directory corrente in forma di puntatore a inode
umask
segnali attivati
segnali inibiti
tempo di utilizzo in quantità di impulsi
stato del processo

eventi attesi
segnale atteso
tempo da attendere
indirizzo efficace delle istruzioni
segmento istruzioni
dimensione istruzioni
indirizzo efficace dei dati
segmento dati
dimensione dati
sp: indice della pila
valore restituito al termine
nome del processo
tabella dei descrittori dei file del processo

Listato 90.81. Struttura del tipo '**proc_t**', corrispondente agli elementi dell'array ***proc_table[]***.

```
typedef struct {
  pid_t             ppid;
  pid_t             pgrp;
  uid_t             uid;
  uid_t             euid;
  uid_t             suid;
  dev_t             device_tty;
  char              path_cwd[PATH_MAX];
  inode_t          *inode_cwd;
  int               umask;
  unsigned long int sig_status;
  unsigned long int sig_ignore;
  clock_t           usage;
  unsigned int      status;
  int               wakeup_events;
  int               wakeup_signal;
  unsigned int      wakeup_timer;
  addr_t            address_i;
  segment_t         segment_i;
  size_t            size_i;
  addr_t            address_d;
  segment_t         segment_d;
  size_t            size_d;
  uint16_t          sp;
  int               ret;
  char              name[PATH_MAX];
  fd_t              fd[FOPEN_MAX];
} proc_t;
```

La tabella successiva descrive il significato dei vari membri previsti dal tipo '**proc_t**'. Va osservato che os16 non gestisce i gruppi di utenti, anche se questi sono previsti comunque nel file system, pertanto la tabella dei processi è più semplice rispetto a quella di un sistema conforme allo standard di Unix. Un'altra considerazione va fatta a proposito della cosiddetta «u-area» (*user area*), la quale non viene gestita come un sistema Unix tradizionale e tutti i dati dei processi sono raccolti nella tabella gestita dal kernel. Di conseguenza, dal momento che i processi non dispongono di una tabella personale con i dati della u-area, devono avvalersi sempre di chiamate di sistema per leggere i dati del proprio processo.

Tabella 90.82. Membri del tipo '**`proc_t`**'.

Membro	Contenuto
`ppid`	Numero del processo genitore: *parent process id*.
`pgrp`	Numero del gruppo di processi a cui appartiene quello della voce corrispondente: *process group*. Si tratta del numero del processo a partire dal quale viene definito il gruppo.
`uid`	Identità reale del processo della voce corrispondente: *user id*. Si tratta del numero dell'utente, secondo la classificazione del file '`/etc/passwd`', per conto del quale il processo è stato avviato. Tuttavia, i privilegi del processo dipendono dall'identità efficace, definita dal membro '**`euid`**'.
`euid`	Identità efficace del processo della voce corrispondente: *effective user id*. Si tratta del numero dell'utente, secondo la classificazione del file '`/etc/passwd`', per conto del quale il processo è in funzione; pertanto, il processo ha i privilegi di quell'utente.
`suid`	Identità salvata: *saved user id*. Si tratta del valore che aveva ***euid*** prima di cambiare identità.
`device_tty`	Terminale di controllo, espresso attraverso il numero del dispositivo.
`path_cwd` `inode_cwd`	Entrambi i membri rappresentano la directory corrente del processo: nel primo caso in forma di percorso, ovvero di stringa, nel secondo in forma di puntatore a inode rappresentato in memoria.
`umask`	Maschera dei permessi associata al processo: i permessi attivi nella maschera vengono tolti in fase di creazione di un file o di una directory.
`sig_status`	Segnali inviati al processo e non ancora trattati: ogni segnale si associa a un bit differente del valore del membro ***sig_status***; un bit a uno indica che il segnale corrispondente è stato ricevuto e non ancora trattato.
`sig_ignore`	Segnali che il processo ignora: ogni segnale da ignorare si associa a un bit differente del valore del membro ***sig_ignore***; un bit a uno indica che quel segnale va ignorato.
`usage`	Tempo di utilizzo della CPU, da parte del processo, espresso in impulsi del temporizzatore, il quale li produce alla frequenza di circa 18,2 Hz.

Membro	Contenuto
`status`	Stato del processo, rappresentabile attraverso una macro-variabile simbolica, definita nel file '`proc.h`'. Per os16, gli stati possibili sono: «inesistente», quando si tratta di una voce libera della tabella dei processi; «creato», quando un processo è appena stato creato; «pronto», quando un processo è pronto per essere eseguito, «in esecuzione», quando il processo è in funzione; «sleeping», quando un processo è in attesa di qualche evento; «zombie», quando un processo si è concluso, ha liberato la memoria, ma rimangono le sue tracce perché il genitore non ha ancora recepito la sua fine.
`wakeup_events`	Eventi attesi per il risveglio del processo, ammesso che si trovi nello stato si attesa. Ogni tipo di evento che può essere atteso corrisponde a un bit e si rappresenta con una macro-variabile simbolica, dichiarata nel file '`lib/sys/os16.h`'.
`wakeup_signal`	Ammesso che il processo sia in attesa di un segnale, questo membro esprime il numero del segnale atteso.
`wakeup_timer`	Ammesso che il processo sia in attesa dello scadere di un conto alla rovescia, questo membro esprime il numero di secondi che devono ancora trascorrere.

`address_i`
`segment_i`
`size_i`
	Il valore di questi membri descrive la memoria utilizzata dal processo per le istruzioni (il segmento codice). Le informazioni sono in parte ridondanti, perché conoscendo ***segment_i*** si ottiene facilmente ***address_i*** e viceversa, ma ciò consente di ridurre i calcoli nelle funzioni che ne fanno uso.

`address_d`
`segment_d`
`size_d`
	Il valore di questi membri descrive la memoria utilizzata dal processo per i dati (il segmento usato per le variabili statiche e per la pila). Anche in questo caso, le informazioni sono in parte ridondanti, ma ciò consente di semplificare il codice nelle funzioni che ne fanno uso.

`sp`	Indice della pila dei dati, nell'ambito del segmento dati del processo. Il valore è significativo quando il processo è nello stato di pronto o di attesa di un evento. Quando invece un processo era attivo e viene interrotto, questo valore viene aggiornato.

Membro	Contenuto
`ret`	Rappresenta il valore restituito da un processo terminato e passato nello stato di «zombie».
`name`	Il nome del processo, rappresentato dal nome del programma avviato.
`fd`	Tabella dei descrittori dei file relativi al processo.

90.8.3 Chiamate di sistema

«

I processi eseguono una chiamata di sistema attraverso la funzione ***sys()***, dichiarata nel file '`lib/sys/os16/sys.s`'. La funzione in sé, per come è dichiarata, potrebbe avere qualunque parametro, ma in pratica ci si attende che il suo prototipo sia il seguente:

```
void sys (syscallnr, void *message, size_t size);
```

Il numero della chiamata di sistema, richiesto come primo parametro, si rappresenta attraverso una macro-variabile simbolica, definita nel file '`lib/sys/os16.h`'.

Per fornire dei dati a quella parte di codice che deve svolgere il compito richiesto, si usa una variabile strutturata, di cui viene trasmesso il puntatore (riferito al segmento dati del processo che esegue la chiamata) e la dimensione complessiva.

Nel file '`lib/sys/os16.h`' sono definiti dei tipi derivati, riferiti a variabili strutturate, per ogni tipo di chiamata. Per esempio, per la chiamata di sistema usata per cambiare la directory corrente del processo, si usa un messaggio di tipo '**`sysmsg_chdir_t`**':

```
typedef struct {
    char          path[PATH_MAX];
    int           ret;
    int           errno;
    int           errln;
    char          errfn[PATH_MAX];
} sysmsg_chdir_t;
```

In realtà, la funzione ***sys()***, si limita a produrre un'interruzione software, da cui viene attivata la routine che inizia al simbolo '**`isr_80`**' nel file '`kernel/_isr.s`', la quale estrapola le informazioni salienti dalla pila dei dati e poi le fornisce alla funzione ***sysroutine()***:

```
void sysroutine (uint16_t *sp, segment_t *segment_d,
                 uint16_t syscallnr, uint16_t msg_off,
                 uint16_t msg_size);
```

Nella funzione ***sysroutine()***, gli ultimi tre parametri corrispondono in pratica agli argomenti della chiamata della funzione ***sys()***, con la differenza che nei vari passaggi hanno perso l'identità originaria e giungono come numeri puri e semplici, secondo la «parola» del tipo di architettura utilizzato.

90.8.4 File «kernel/proc/...»

Listati successivi a 100.9.

Nella directory '`kernel/proc/`' si trovano i file che realizzano le funzioni dichiarate all'interno di '`kernel/proc.h`'.

Nella gestione dei processi entrano in gioco due variabili globali importanti: ***_ksp*** e ***_etext***. La prima è dichiarata nel file '`kernel/_isr.s`' e viene utilizzata per annotare l'indice della pila dei dati del kernel; la seconda è dichiarata implicitamente dal collegatore (*linker*) e contiene la dimensione dell'area occupata in memoria dal codice del kernel stesso.

Nel file '`kernel/proc/proc_table.c`' è dichiarata la tabella dei processi, attraverso un array composto da elementi di tipo '**`proc_t`**'. La quantità di elementi di questo array costituisce il limite alla quantità di processi gestibili simultaneamente, incluso il kernel e i processi zombie.

Per accedere uniformemente al contenuto della tabella, si usa la funzione ***proc_reference()***, la quale, con l'indicazione del numero del processo (PID), restituisce il puntatore all'elemento della tabella che contiene i dati dello stesso.

Nelle sezioni successive si descrivono solo le funzioni principali della directory '`kernel/proc/`'.

90.8.4.1 Funzione «proc_init()»

```
void proc_init (void);
```

La funzione ***proc_init()*** viene chiamata dalla funzione ***main()***, una volta sola, per attivare la gestione dei processi elaborativi. Si occupa di compiere le azioni seguenti:

- modificare la tabella delle interruzioni (IVT), attraverso la chiamata della funzione ***_ivt_load()*** (per comodità si usa la macro-istruzione ***ivt_load()***), dichiarata nel file '`kernel/proc/_ivt_load.s`';
- impostare la frequenza del temporizzatore, ma tale frequenza deve essere obbligatoriamente di 18,2 Hz;
- azzerare la tabella dei processi;
- innestare il file system principale;
- assegnare i valori appropriati alla voce della tabella dei processi che si riferisce al kernel (PID zero);
- allocare la memoria già utilizzata dal kernel e lo spazio che va da zero fino a 00500_{16} (tabella IVT e BDA);
- attivare selettivamente le interruzioni hardware desiderate.

90.8.4.2 Funzione «sysroutine()»

«

La funzione ***sysroutine()*** viene chiamata esclusivamente dalla routine attivata dalle chiamate di sistema (tale routine è introdotta dal simbolo '**`isr_80`**' nel file '`kernel/proc/_isr.s`') e ha una serie di parametri, come si può vedere dal prototipo:

```
void sysroutine (uint16_t *sp, segment_t *segment_d,
                 uint16_t syscallnr, uint16_t msg_off,
                 uint16_t msg_size);
```

I primi due parametri della funzione sono puntatori a variabili dichiarate nel file '`kernel/proc/_isr.s`'. La prima delle due variabili è l'indice della pila dei dati del processo che ha eseguito la chiamata di sistema; la seconda contiene l'indirizzo del segmento dati di tale processo. Il valore del segmento dati serve a individuare il processo elaborativo nella tabella dei processi, dal momento che con os16 i dati non sono condivisibili tra processi.

Il terzo parametro è il numero della chiamata di sistema che ha provocato l'interruzione. Gli ultimi due parametri danno la posizione e la dimensione del messaggio inviato attraverso la chiamata di sistema.

All'inizio della funzione viene individuato il processo elaborativo corrispondente a quello che utilizza il segmento dati ****segment_d*** e l'indirizzo efficace dell'area di memoria contenente il messaggio della chiamata di sistema:

```
    pid_t  pid       = proc_find (*segment_d);
    addr_t msg_addr = address (*segment_d, msg_off);
```

Quindi viene dichiarata un'unione di variabili strutturate, corrispondente alla sovrapposizione di tutti i tipi di messaggio gestibili:

```
    union {
        sysmsg_chdir_t     chdir;
        sysmsg_chmod_t     chmod;
        ...
    } msg;
```

A questo punto si verifica se il processo interrotto dalla sua chiamata di sistema è il kernel, perché al kernel è consentito di eseguire solo alcuni tipi di chiamata e tutto il resto sarebbe un errore.

Proseguendo con il codice si vede l'uso della funzione ***dev_io()***, con la quale si legge il messaggio della chiamata di sistema, dalla sua collocazione originale, in un'area tampone del segmento dati del kernel:

```
    dev_io (pid, DEV_MEM, DEV_READ, msg_addr, &msg,
            msg_size, NULL);
```

A questo punto, sapendo di quale chiamata di sistema si tratta, il messaggio può essere letto come:

```
msg.tipo_chiamata
```

Per esempio, per la chiamata di sistema '**SYS_CHDIR**', si deve fare riferimento al messaggio ***msg.chdir***; pertanto, per raggiungere il membro ***ret*** del messaggio si usa la notazione ***msg.chdir.ret***.

Una volta eseguita una copia del messaggio, con la funzione ***dev_io()***, si passa a una struttura di selezione, con cui si eseguono operazioni differenti in base al tipo di chiamata ricevuta:

```
    switch (syscallnr)
      {
        case SYS_0:
            break;
        case SYS_CHDIR:
            msg.chdir.ret = path_chdir (pid, msg.chdir.path);
            sysroutine_error_back (&msg.chdir.errno,
                                   &msg.chdir.errln,
                                   msg.chdir.errfn);
            break;
        ...
      }
```

Il messaggio usato per trasmettere i dati della chiamata, può servire anche per restituire dei dati al mittente, pertanto, spesso alcuni contenuti dello stesso vengono modificati. Ciò succede particolarmente con il membro ***ret*** che generalmente rappresenta il valore restituito dalla chiamata di sistema. Per questa ragione, dopo la struttura di selezione si ricopia nuovamente il messaggio nella posizione di partenza:

```
    dev_io (pid, DEV_MEM, DEV_WRITE, msg_addr, &msg,
            msg_size, NULL);
```

Al termine del lavoro, viene chiamata la funzione ***proc_scheduler()***.

90.8.4.3 Funzione «proc_scheduler()»

«

La funzione ***proc_scheduler()*** richiede come parametri due puntatori: il primo parametro deve essere il riferimento a un valore che rappresenta l'indice della pila di quel processo; il secondo parametro si riferisce a una variabile contenente il valore del segmento dati del processo interrotto. La funzione richiede queste informazioni in forma di puntatore, per poter modificare i valori delle variabili relative, in modo da consentire così l'attivazione successiva di un altro processo, al posto di quello da cui si proviene.

```
void proc_scheduler (uint16_t *sp, segment_t *segment_d);
```

Inizialmente, la funzione acquisisce il numero del processo interrotto:

```
    prev = proc_find (*segment_d);
```

Quindi svolge delle operazioni che riguardano tutti i processi: aggiorna i contatori dei processi che attendono lo scadere di un certo tempo; verifica la presenza di segnali e predispone le azioni relative; raccoglie l'input dai terminali.

```
    proc_sch_timers ();
    …
    proc_sch_signals ();
    …
    proc_sch_terminals ();
```

A quel punto aggiorna il tempo di utilizzo della CPU del processo appena interrotto:

```
    current_clock   = k_clock ();
    ps[prev].usage += current_clock - previous_clock;
    previous_clock  = current_clock;
```

Quindi inizia la ricerca di un altro processo, candidato a essere ripreso, al posto di quello interrotto. La ricerca inizia dal processo successivo a quello interrotto, senza considerare alcun criterio di precedenza. Il ciclo termina se la ricerca incontra di nuovo il processo di partenza.

```
    for (next = prev+1; next != prev; next++)
      {
        if (next >= PROCESS_MAX)
          {
            next = -1; // At the next loop, 'next' will be
                       // zero.
            continue;
          }
        …
      }
```

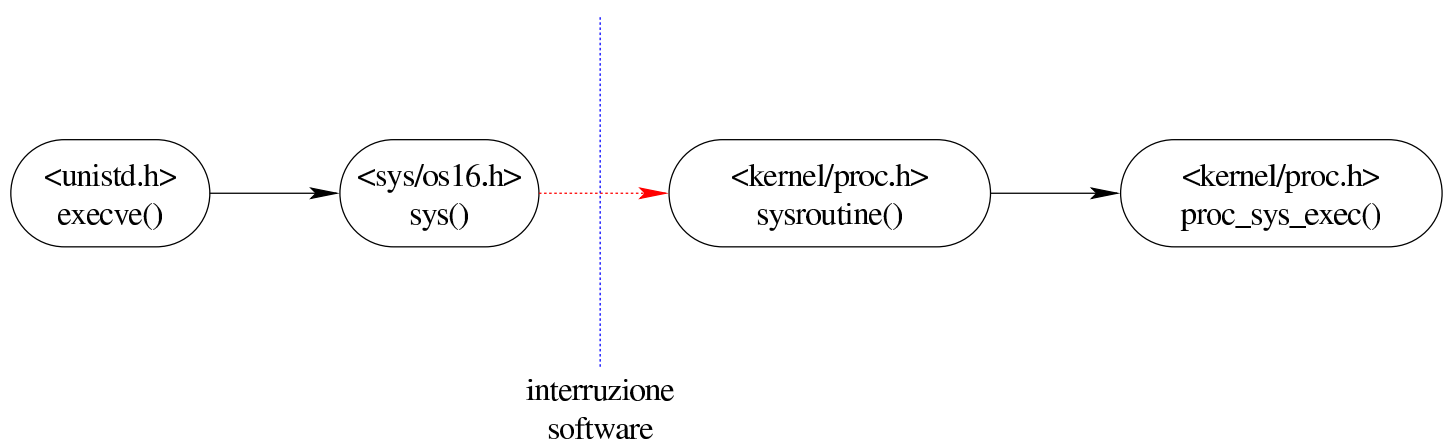
All'interno di questo ciclo di ricerca, se si incontra un processo pronto per essere messo in funzione, lo si scambia con quello interrotto: in pratica si salva il valore attuale dell'indice della pila, si scambiano gli stati e si aggiornano i valori di ***sp** e ***segment_d**, in modo da ottenere effettivamente lo scambio all'uscita dalla funzione:

```
        else if (ps[next].status == PROC_READY)
          {
            if (ps[prev].status == PROC_RUNNING)
              {
                ps[prev].status = PROC_READY;
              }
            ps[prev].sp     = *sp;
            ps[next].status = PROC_RUNNING;
            ps[next].ret    = 0;
            *segment_d      = ps[next].segment_d;
            *sp             = ps[next].sp;
            break;
          }
```

Alla fine del ciclo, occorre verificare se esiste effettivamente un processo successivo attivato, perché in caso contrario, si lascia il controllo direttamente al kernel. In fine, si salva il valore accumulato in precedenza dell'indice della pila del kernel, nella variabile ***_ksp***, quindi si manda il messaggio EOI al circuito del PIC (*programmable interrupt controller*), diversamente non ci sarebbero più, altre interruzioni.

90.9 Caricamento ed esecuzione delle applicazioni

Caricare un programma e metterlo in esecuzione è un processo delicato che parte dalla funzione ***execve()*** della libreria standard e viene svolto dalla funzione ***proc_sys_exec()*** del kernel.

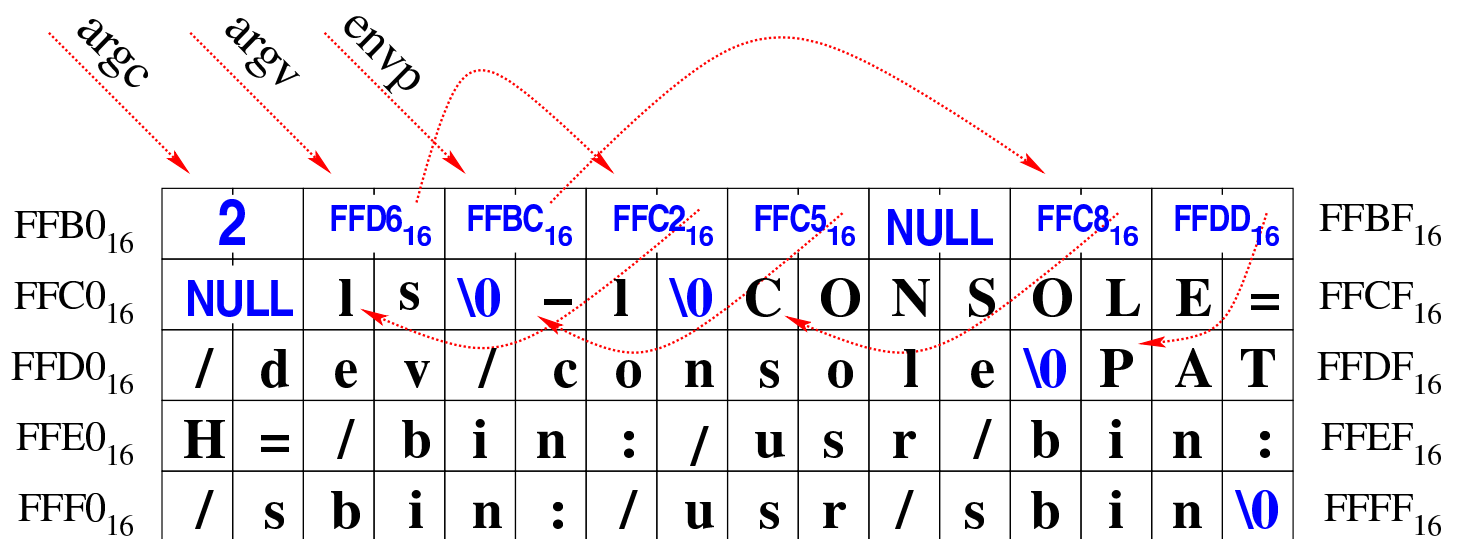
Figura 90.94. Da ***execve()*** a ***proc_sys_exec()***.

90.9.1 Caricamento in memoria

La funzione ***proc_sys_exec()*** (listato 100.9.21) del kernel è quella che svolge il compito di caricare un processo in memoria e di annotarlo nella tabella dei processi.

La funzione, dopo aver verificato che si tratti di un file eseguibile valido e che ci siano i permessi per metterlo in funzione, procede all'allocazione della memoria, dividendo se necessario l'area codice da quella dei dati, quindi legge il file e copia opportunamente le componenti di questo nelle aree di memoria allocate.

> La realizzazione attuale della funzione ***proc_sys_exec()*** non è in grado di verificare se un processo uguale sia già in memoria, quindi carica la parte del codice anche se questa potrebbe essere già disponibile.

Terminato il caricamento del file, viene ricostruita in memoria la pila dei dati del processo. Prima si mettono sul fondo le stringhe delle variabili di ambiente e quelle degli argomenti della chiamata, quindi si aggiungono i puntatori alle stringhe delle variabili di ambiente, ricostruendo così l'array noto convenzionalmente come '**`envp[]`**', continuando con l'aggiunta dei puntatori alle stringhe degli argomenti della chiamata, per riprodurre l'array '**`argv[]`**'. Per ricostruire gli argomenti della chiamata della funzione ***main()*** dell'applicazione, vanno però aggiunti ancora: il puntatore all'inizio dell'array delle stringhe che descrivono le variabili di ambiente, il puntatore all'array delle stringhe che descrivono gli argomenti della chiamata e il valore che rappresenta la quantità di argomenti della chiamata.

Figura 90.95. Caricamento degli argomenti della chiamata della funzione ***main()***.

argc argv envp

$FFB0_{16}$	2		$FFD6_{16}$		$FFBC_{16}$		$FFC2_{16}$		$FFC5_{16}$		NULL		$FFC8_{16}$		$FFDD_{16}$		$FFBF_{16}$
$FFC0_{16}$	NULL		l	s	\0	–	l	\0	C	O	N	S	O	L	E	=	$FFCF_{16}$
$FFD0_{16}$	/	d	e	v	/	c	o	n	s	o	l	e	\0	P	A	T	$FFDF_{16}$
$FFE0_{16}$	H	=	/	b	i	n	:	/	u	s	r	/	b	i	n	:	$FFEF_{16}$
$FFF0_{16}$	/	s	b	i	n	:	/	u	s	r	/	s	b	i	n	\0	$FFFF_{16}$

Fatto ciò, vanno aggiunti tutti i valori necessari allo scambio dei processi, costituiti dai vari registri da rimpiazzare.

Figura 90.96. Completamento della pila con i valori dei registri.

FF90$_{16}$									AX		CX		DX		BX		FF9F$_{16}$
FFA0$_{16}$	BP		SI		DI		DS		ES		IP		CS		FLAGS		FFAF$_{16}$
FFB0$_{16}$	2		FFD6$_{16}$		FFBC$_{16}$		FFC2$_{16}$		FFC5$_{16}$		NULL		FFC8$_{16}$		FFDD$_{16}$		FFBF$_{16}$
FFC0$_{16}$	NULL		l	s	\0	-	l	\0	C	O	N	S	O	L	E	=	FFCF$_{16}$
FFD0$_{16}$	/	d	e	v	/	c	o	n	s	o	l	e	\0	P	A	T	FFDF$_{16}$
FFE0$_{16}$	H	=	/	b	i	n	:	/	u	s	r	/	b	i	n	:	FFEF$_{16}$
FFF0$_{16}$	/	s	b	i	n	:	/	u	s	r	/	s	b	i	n	\0	FFFF$_{16}$

Superato il problema della ricostruzione della pila dei dati, la funzione ***proc_sys_exec()*** predispone i descrittori di standard input, standard output e standard error, quindi libera la memoria usata dal processo chiamante e ne rimpiazza i dati nella tabella dei processi con quelli del nuovo processo caricato.

90.9.2 Il codice iniziale dell'applicativo

I programmi iniziano con il codice che si trova nel file '`applic/crt0.s`'. Questo file ha delle affinità con il file '`kernel/main/crt0.s`' del kernel, dove la prima differenza che si incontra riguarda l'impronta di riconoscimento. A parte questo, va considerato che il codice delle applicazioni viene eseguito in un momento in cui i registri di segmento sono già stati impostati e l'indice della pila è già collocato correttamente; inoltre, se la funzione ***main()*** termina e restituisce il controllo a '`crt0.s`', un ciclo senza fine esegue continuamente una chiamata di sistema per la conclusione del processo elaborativo corrispondente.

Figura 90.97. Codice iniziale degli applicativi e variabile strutturata di tipo '**header_t**'.

```
entry startup
.text
startup:
    jmp startup_code
filler:
    .space (0x0004 - (filler - startup))
magic:
    .data4 0x6F733136
    .data4 0x6170706C
segoff:
    .data2 __segoff
etext:
    .data2 __etext
edata:
    .data2 __edata
ebss:
    .data2 __end
stack_size:
    .data2 0x2000
.align 2
startup_code:
…
```

```
typedef struct {
    uint32_t filler0;
    uint32_t magic0;
    uint32_t magic1;
    uint16_t segoff;
    uint16_t etext;
    uint16_t edata;
    uint16_t ebss;
    uint16_t ssize;
} header_t;
```

La figura mostra il confronto tra il codice iniziale contenuto nel file '`applic/crt0.s`', senza preamboli e senza commenti, con la dichiarazione del tipo derivato '**header_t**', presente nel file '`kernel/proc.h`'. Attraverso questa struttura, la funzione ***proc_sys_exec()*** è in grado di estrapolare dal file le informazioni necessarie a caricarlo correttamente in memoria.

Come già accennato, quando viene eseguito il codice di un programma applicativo, la pila dei dati è già operativa. Pertanto, dopo il simbolo '**startup_code**' si può già lavorare con questa.

```
startup_code:
    pop ax              ; argc
    pop bx              ; argv
    pop cx              ; envp
    mov _environ, cx    ; Variable 'environ' comes from
                        ; 'unistd.h'.
    push cx
    push bx
    push ax
```

Per prima cosa, viene estratto dalla pila il puntatore all'array noto come ***envp[]***, per poter assegnare tale valore alla variabile ***environ***, come richiede lo standard della libreria POSIX. Tuttavia, per poter gestire poi le variabili di ambiente, si rende necessario utilizzare un array più «comodo», quando le stringhe vanno sostituite. A tale proposito, nel file '`lib/stdlib/environment.c`', si dichiarano ***_environment_table[][]*** e ***_environment[]***. Il primo è semplicemente un array di caratteri, dove, utilizzando due indici di accesso, si conviene di allocare delle stringhe, con una dimensione massima prestabilita. Il secondo, invece, è un array di puntatori, per localizzare l'inizio delle stringhe contenute nel primo. In pratica, alla fine ***_environment[]*** e ***environ[]*** devono essere equivalenti. Ma per attuare questo, occorre utilizzare la funzione ***_environment_setup()*** che sistema tutti i puntatori necessari.

```
    push cx
    call __environment_setup
    add  sp, #2
    ;
    mov  ax, #__environment
    mov  _environ, ax
    ;
    pop ax              ; argc
    pop bx              ; argv[][]
    pop cx              ; envp[][]
    mov cx, #__environment
    push cx
    push bx
    push ax
```

Come si vede dall'estratto del file '`applic/crt0.s`', si vede l'uso della funzione ***_environment_setup()*** (il registro CX contiene già il puntatore a ***envp[]***, e viene

inserito nella pila proprio come argomento per la funzione). Successivamente viene riassegnata anche la variabile ***environ*** in modo da coincidere con ***_environment***. Alla fine, viene ricostruita la pila per gli argomenti della chiamata della funzione ***main()***, ma prima di procedere con quella chiamata, si utilizzano due funzioni, per inizializzare la gestione dei flussi di file e delle directory, sempre in forma di flussi.

```
    call __stdio_stream_setup
    call __dirent_directory_stream_setup
    ;
    call _main
    ;
    mov  exit_value, ax
    …
.align 2
.data
exit_value:
    .data2 0x0000
.align 2
.bss
```

La funzione ***_stdio_stream_setup()***, contenuta nel file '`lib/stdio/FILE.c`', associa i descrittori standard ai flussi di file standard (standard input, standard output e standard error); la funzione ***_dirent_directory_stream_setup()*** compie un lavoro analogo, limitandosi però a inizializzare un array di flussi di directory.

Dopo queste preparazioni, viene chiamata la funzione ***main()***, la quale riceve regolarmente i propri argomenti previsti. Il valore restituito dalla funzione viene poi salvato in corrispondenza del simbolo '**`exit_value`**'.

```
halt:
    push #2             ; Size of message.
    push #exit_value    ; Pointer to the message.
    push #6             ; SYS_EXIT
    call _sys
    add  sp, #2
    add  sp, #2
    add  sp, #2
    jmp halt
```

All'uscita dalla funzione ***main()***, dopo aver salvato quanto restituito dalla funzione stessa, ci si introduce nel codice successivo al simbolo '**`halt`**', nel quale si chiama la

funzione ***sys()*** (chiamata di sistema), per produrre la chiusura formale del processo. Ciò che si vede è comunque l’equivalente di ‘**`_exit (`*exit_status*`);`**’.[1]

[1] Va tenuto in considerazione che ‘**`exit_status`**’ è un simbolo non raggiungibile dal codice C, perché dovrebbe essere esportato con un nome che inizi con il trattino basso.

Parte vi

Manuale

Avvio del sistema e conclusione

TXT
htm
PDF
pdf
P1
P2
P3
P4
epub

os16 ha due modalità di funzionamento: si può interagire direttamente con il kernel, oppure si può avviare il processo '**`init`**' e procedere con l'organizzazione consueta di un sistema Unix tradizionale. La modalità di colloquio diretto con il kernel è servita per consentire lo sviluppo di os16 e potrebbe essere utile per motivi di studio. Va osservato che durante l'interazione diretta con il kernel si dispone di una sola console, mentre quando si avvia '**`init`**' si possono avere delle console virtuali in base alla configurazione.

91.1 Interazione con il kernel

L'avvio di os16 passa, in ogni caso, per una prima fase di colloquio con il kernel. Si ottiene un menù e si possono premere semplicemente dei tasti, secondo l'elenco previsto, per ottenere delle azioni molto semplici. In questa fase il disco da cui risulta avviato il kernel è già innestato ed è prevista la possibilità di avviare tre programmi: '`/bin/aaa`', '`/bin/bbb`' e '`/bin/ccc`'. In tal modo, si ha la possibilità di avviare qualcosa, a titolo diagnostico, prima dello stesso '**`init`**' ('`/bin/init`'.

Figura 91.1. Aspetto di os16 in funzione, con il menù in evidenza.

```
os16 build 20AA.MM.GG HH:MM:SS ram 639 Kibyte
.------------------------------------------------------------------.
| [h]       show this menu                                         |
| [p]       process status and memory map                          |
| [1]..[9] kill process  1 to 9                                    |
| [A]..[F] kill process 10 to 15                                   |
| [l]       send SIGCHLD to process 1                              |
| [a]..[c] run programs '/bin/aaa' to '/bin/ccc' in parallel       |
| [f]       system file status                                     |
| [n], [N] list of active inodes                                   |
| [z]       print root file system zone map (read left to right)   |
| [m], [M] mount/umount '/dev/dsk1' at '/usr/'                     |
| [x]       exit interaction with kernel and start '/bin/init'     |
| [q]       quit kernel                                            |
`------------------------------------------------------------------'
```

Le funzioni principali disponibili in questa modalità diagnostica sono riassunte nella tabella successiva:

Tasto	Risultato
[*h*]	Mostra il menù di funzioni disponibili.
[*1*]	Invia il segnale '**`SIGKILL`**' al processo numero uno.
[*2*]...[*9*]	Invia il segnale '**`SIGTERM`**' al processo con il numero corrispondente.

Tasto	Risultato
[A]...[F]	Invia il segnale '**`SIGTERM`**' al processo con il numero da 10 a 15.
[a], [b], [c]	Avvia il programma '`/bin/aaa`', '`/bin/bbb`' o '`/bin/ccc`'.
[f]	Mostra l'elenco dei file aperti nel sistema.
[m], [M]	Innesta o stacca il secondo dischetto dalla directory '`/usr/`'.
[n], [N]	Mostra l'elenco degli inode aperti: l'elenco è composto da due parti.
[l]	Invia il segnale '**`SIGCHLD`**' al processo numero uno.
[p]	Mostra la situazione dei processi e altre informazioni.
[x]	Termina il ciclo e successivamente si passa all'avvio di '`/bin/init`'.
[q]	Ferma il sistema.

Figura 91.3. Aspetto di os16 in funzione mentre visualizza anche la tabella dei processi avviati (tasto [p]).

```
ababaaababaaababaabbaaababaaababap
pp  p pg
id id rp  tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name
 0  0  0 0000    0    0    0 00.35 R 10500 eb7c 00500 0000 ffc8 os16 kernel
 0  1  0 0000    0    0    0 00.33 r 2f100 0600 2f700 aa00 a8e8 /bin/ccc
 0  2  0 0000    0    0    0 00.01 r 1f100 0600 84300 aa00 a8e8 /bin/ccc
 2  3  0 0000   10   10   10 00.01 r 1f100 0600 44b00 aa00 a8e8 /bin/ccc
 0  4  0 0000    0    0    0 00.17 r 21600 0600 3a100 aa00 a8e8 /bin/ccc
 4  5  0 0000   10   10   10 00.02 r 21c00 2400 6f100 a900 a86c /bin/aaa
 4  6  0 0000   11   11   11 00.02 s 24000 2500 59d00 a900 a8b6 /bin/bbb
 0  7  0 0000    0    0    0 00.13 r 26500 0600 64600 aa00 a8e8 /bin/ccc
 7  8  0 0000   10   10   10 00.02 r 26b00 2400 8ee00 a900 981e /bin/aaa
 7  9  0 0000   11   11   11 00.02 s 2bf00 2500 79a00 a900 a8b6 /bin/bbb
CS=1050 DS=0050 SS=0050 ES=0050 BP=ffe4 SP=ffe4 heap_min=878c etext=eb7c edata=1
b3c ebss=4c34 ksp=ffc8 clock=0000084b, time elapsed=00:01:57
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffe00000003ffffffffffffffffffffffffff
fffe000000000001fffffffff0007fffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff800
000000000000000000000000000000000000000007ffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffff7fffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffbfffffffffffffffffffffffffffffffffffffffffe0000000000000000000000000f

abbaaaaababaaababaaabbaabbaababab baabbbbbbbbbbbbb
```

Premendo [x], il ciclo termina e il kernel avvia '`/bin/init`', ma prima di farlo occorre che non ci siano altri processi in funzione, perché '**`init`**' deve assumere il ruolo di processo 1, ovvero il primo dopo il kernel.

Figura 91.4. Aspetto di os16 in funzione con il menù in evidenza, dopo aver premuto il tasto [x] per avviare '**init**'.

```
os16 build 20AA.MM.GG HH:MM:SS ram 639 Kibyte
.--------------------------------------------------------------------.
| [h]       show this menu                                           |
| [p]       process status and memory map                            |
| [1]..[9] kill process  1 to 9                                      |
| [A]..[F] kill process 10 to 15                                     |
| [l]       send SIGCHLD to process 1                                |
| [a]..[c] run programs '/bin/aaa' to '/bin/ccc' in parallel        |
| [f]       system file status                                       |
| [n], [N] list of active inodes                                     |
| [z]       print root file system zone map (read left to right)     |
| [m], [M] mount/umount '/dev/dsk1' at '/usr/'                       |
| [x]       exit interaction with kernel and start '/bin/init'       |
| [q]       quit kernel                                              |
`--------------------------------------------------------------------'
init
os16: a basic os. [Ctrl q], [Ctrl r], [Ctrl s], [Ctrl t] to change
console.
This is terminal /dev/console0
Log in as "root" or "user" with password "ciao" :-)
login:
```

91.2 Avvio e conclusione del sistema «normale»

Se non si intende operare direttamente con il kernel, come descritto nella sezione precedente, con la pressione del tasto [x] si avvia '**init**'.

Il programma '**init**' legge il file '`/etc/inittab`' e sulla base del suo contenuto, avvia uno o più processi '**getty**', per la gestione dei vari terminali disponibili (si tratta comunque soltanto di console virtuali).

Il programma '**getty**' apre il terminale che gli viene indicato come opzione della chiamata (da '**init**' che lo determina in base al contenuto di '`/etc/inittab`'), facendo in modo che sia associato al descrittore zero (standard input). Quindi, dopo aver visualizzato il contenuto del file '`/etc/issue`', mostra un proprio messaggio e avvia il programma '**login**'.

Il programma '**login**' prende il posto di '**getty**' che così scompare dall'elenco dei processi. '**login**' procede chiedendo all'utente di identificarsi, utilizzando il file '`/etc/passwd`' per verificare le credenziali di accesso. Se l'identificazione ha successo, viene avviata la shell definita nel file '`/etc/passwd`' per l'utente, in modo

da sostituirsi al programma '**`login`**', il quale scompare a sua volta dall'elenco dei processi.

Attraverso la shell è possibile interagire con il sistema operativo, secondo la modalità «normale», nei limiti delle possibilità di os16. Quando la shell termina di funzionare, '**`init`**' riavvia '**`getty`**'.

Per cambiare console virtuale si possono usare le combinazioni [*Ctrl q*], [*Ctrl r*], [*Ctrl s*] e [*Ctrl t*], ma bisogna considerare che dipende dalla configurazione del file '`/etc/inittab`' se effettivamente vengono attivate tutte queste console.

Per concludere l'attività del sistema, basta concludere il funzionamento delle varie sessioni di lavoro (la shell finisce di funzionare con il comando interno '**`exit`**') e non c'è bisogno di altro; pertanto, non è previsto l'uso di comandi come '**`halt`**' o '**`shutdown`**' e, d'altro canto, le operazioni di scrittura nel file system sono sincrone, in modo tale da non richiedere accorgimenti particolari per la chiusura delle attività.

Riquadro 91.5. Coerenza del file system.

> os16 è destinato a essere usato attraverso un emulatore, dove gli unici due dischi previsti (dischetti da 1440 Kibyte) sono semplicemente dei file. Pertanto, la manutenzione del file system, avviene all'esterno di os16, di norma con un sistema GNU/Linux. Tuttavia, occorre tenere presente che quando si usa il sistema os16, tramite l'emulatore, i file-immagine dei dischi di os16 non devono essere innestati nel sistema operativo ospitante, perché altrimenti le operazioni di scrittura eseguite da os16 potrebbero essere annullate, anche solo parzialmente, dalla gestione del sistema ospitante, rendendo il file system incoerente.

Capitolo 92

Sezione 1: programmi eseguibili o comandi interni di shell

TXT
htm
PDF
pdf
P1
P2
P3
P4
epub

92.1 os16: aaa(1)

NOME

'**aaa**', '**bbb**', '**ccc**' - programmi elementari avviabili direttamente dal kernel

SINTASSI

```
aaa
```

```
bbb
```

```
ccc
```

DESCRIZIONE

'**aaa**' e '**bbb**' si limitano a visualizzare una lettera, rispettivamente «a» e «b», attraverso lo standard output, a intervalli regolari. Precisamente, '**aaa**' lo fa a intervalli di un secondo, mentre '**bbb**' a intervalli di due secondi. Il lavoro di '**aaa**' e di '**bbb**' si conclude dopo l'emissione, rispettivamente, di 60 e 30 caratteri, pertanto nel giro di un minuto di tempo si esaurisce il loro compito.

Il programma '**ccc**' è diverso, ma nasce per lo stesso scopo: controllare la gestione dei processi di os16. Questo programma si limita ad avviare, '**aaa**' e '**bbb**', come propri processi-figli, rimanendo in funzione, senza fare nulla. Pertanto, se si usa '**ccc**', il suo processo deve essere eliminato in modo esplicito, perché da solo non si concluderebbe mai.

Questi programmi sono indicati soprattutto per l'uso di os16 nella modalità interattiva che precede il funzionamento normale del sistema operativo, per la verifica della gestione dei processi.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/aaa.c`' [102.1.2]

'`applic/bbb.c`' [102.1.3]

'`applic/ccc.c`' [102.1.5]

92.2 os16: bbb(1)

«

Vedere *aaa(1)* [92.1].

92.3 os16: cat(1)

«

NOME

'**cat**' - emissione del contenuto di uno o più file attraverso lo standard output

SINTASSI

```
cat [file]...
```

DESCRIZIONE

'**cat**' legge il contenuto dei file indicati come argomento e li emette attraverso lo standard output, concatenati assieme in un unico flusso.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/cat.c' [102.1.4]

VEDERE ANCHE

more(1) [92.16], *ed(1)* [92.9].

92.4 os16: ccc(1)

«

Vedere *aaa(1)* [92.1].

92.5 os16: chmod(1)

«

NOME

'**chmod**' - cambiamento della modalità dei permessi dei file

SINTASSI

```
chmod mod_ottale file...
```

DESCRIZIONE

'**chmod**' cambia la modalità dei permessi associati ai file indicati, in base al numero ottale indicato come primo argomento.

NOTE

Questa versione di '**chmod**' non permette di indicare la modalità dei permessi in forma simbolica.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/chmod.c`' [102.1.6]

VEDERE ANCHE

chown(1) [92.6].

92.6 os16: chown(1)

«

NOME

'**chown**' - cambiamento del proprietario di un file

SINTASSI

chown ***nome_utente file***...

chown ***uid file***...

DESCRIZIONE

'**chown**' cambia l'utente proprietario dei file indicati. Il nuovo proprietario da attribuire può essere indicato per nome o per numero.

NOTE

os16 non gestisce i gruppi, pertanto si può intervenire soltanto sull'utente proprietario dei file.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/chown.c`' [102.1.7]

VEDERE ANCHE

chmod(1) [92.5].

92.7 os16: cp(1)

NOME

'**cp**' - copia dei file

SINTASSI

```
cp file_orig file_nuovo...
```

```
cp file... directory_dest...
```

DESCRIZIONE

'**cp**' copia uno o più file. Se l'ultimo argomento è costituito da una directory esistente, la copia produce dei file con lo stesso nome degli originali, all'interno della directory; se l'ultimo argomento non è una directory già esistente, ci può essere un solo file da copiare, intendendo che si voglia creare una copia con quel nome specificato.

DIFETTI

Non è possibile copiare oggetti diversi dai file puri e semplici; quindi, la copia ricorsiva di una directory non è ammissibile.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/cp.c`' [102.1.8]

VEDERE ANCHE

touch(1) [92.20], *mkdir(1)* [92.15].

92.8 os16: date(1)

NOME

'**date**' - visualizzazione o impostazione della data e dell'ora di sistema

SINTASSI

```
date [MMGGhhmm [[SS] AA]]
```

DESCRIZIONE

Se si utilizza il programma '**date**' senza argomenti, si ottiene la visualizzazione della data e dell'ora attuale del sistema operativo. Se si indica una sequenza numerica come argomento, si intende invece impostare la data e l'ora del sistema. In tal caso va indicato un numero preciso di cifre, che può essere di otto, dieci o dodici. Se si immettono otto cifre, si sta specificando il mese, il giorno, l'ora e i minuti dell'anno attuale; se si indicano dieci cifre, le ultime due rappresentano l'anno del secolo attuale; se si immettono dodici cifre, l'anno è indicato per esteso nelle ultime quattro cifre.

ESEMPI

`# date 123123592012` [*Invio*]

Imposta la data di sistema al giorno 31 dicembre 2012, alle ore 23:59.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/date.c`' [102.1.10]

VEDERE ANCHE

time(2) [93.39], *stime(2)* [93.39].

92.9 os16: ed(1)

NOME

'**ed**' - creazione e modifica di file di testo

SINTASSI

```
ed
```

DESCRIZIONE

'**ed**' è un programma di creazione e modifica di file di testo, che consente di operare su una riga alla volta.

'**ed**' opera in due modalità di funzionamento: comando e inserimento. All'avvio, '**ed**' si trova in modalità di comando, durante la quale ciò che si inserisce attraverso lo standard input viene interpretato come un comando da eseguire. Per esempio, il comando '**1i**' richiede di passare alla modalità di inserimento, immettendo delle righe a partire dalla prima posizione, spostando quelle presenti

in basso. Quando ci si trova in modalità di inserimento, per poter passare alla modalità di comando si introduce un punto isolato, all'inizio di una nuova riga.

Per il momento, in questa pagina di manuale, si omette la descrizione completa dell'utilizzo di '**`ed`**'.

DIFETTI

La digitazione da tastiera viene interpretata da '**`ed`**' in modo letterale. Pertanto, anche la cancellazione, [*Backspace*], benché visivamente faccia indietreggiare il cursore, in realtà introduce il codice *<DEL>*. In fase di inserimento ciò comporta la scrittura di tale codice; in modalità di comando, ciò rende errato l'inserimento.

Il file che si intende elaborare con '**`ed`**' viene caricato o creato completamente nella memoria centrale. Dal momento che os16 consente a ogni processo di gestire una quantità molto limitata di memoria, si può lavorare soltanto con file di dimensioni estremamente ridotte.

AUTORI

Questa edizione di '**`ed`**' è stata scritta originariamente da David I. Bell, per '**`sash`**' (una shell che include varie funzionalità, da compilare in modo statico). Successivamente, il codice è stato estrapolato da '**`sash`**' e reso indipendente, per gli scopi del sistema operativo ELKS (una versione a 16 bit di Linux). Dalla versione estratta per ELKS è stata ottenuta quella di os16, con una serie di modifiche apportate da Daniele Giacomini, tra cui risulta particolarmente evidente il cambiamento dello stile di impaginazione del codice.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/ed.c`' [102.1.11]

VEDERE ANCHE

shell(1) [92.19].

92.10 os16: kill(1)

«

NOME

'**`kill`**' - invio di un segnale ai processi

SINTASSI

```
kill -s nome_segnale pid...
```

```
kill -l
```

DESCRIZIONE

Il programma '**kill**' consente di inviare un segnale, indicato per nome, a uno o più processi, specificati per numero.

OPZIONI

Opzione	Descrizione
-l	Mostra l'elenco dei nomi dei segnali disponibili.
-s ***nome_segnale***	Specifica il nome del segnale da inviare ai processi.

NOTE

Non è possibile indicare il segnale per numero, perché lo standard definisce i nomi di un insieme di segnali necessari, ma non stabilisce il numero, il quale può essere attribuito liberamente in fase realizzativa.

DIFETTI

os16 non consente ai processi di attribuire azioni alternative ai segnali; pertanto, si possono ottenere solo quelle predefinite. Tutto quello che si può fare è, eventualmente, bloccare i segnali, esclusi però quelli che non sono mascherabili per loro natura.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/kill.c' [102.1.14]

VEDERE ANCHE

kill(2) [93.22], *signal(2)* [93.34].

92.11 os16: ln(1)

NOME

'**ln**' - collegamento dei file

SINTASSI

```
ln file_orig file_nuovo...
```

```
ln file... directory_dest...
```

DESCRIZIONE

'**ln**' crea il collegamento fisico di uno o più file. Se l'ultimo argomento è costituito da una directory esistente, si producono collegamenti con gli stessi nomi degli originali, all'interno della directory; se l'ultimo argomento non è una directory già esistente, ci può essere un solo file da collegare, intendendo che si voglia creare un collegamento con quel nome specificato.

Essendo disponibile soltanto la creazione di collegamenti fisici, questi collegamenti possono essere collocati soltanto all'interno del file system di quelli originali, senza contare eventuali innesti ulteriori.

DIFETTI

Non è possibile creare dei collegamenti simbolici, perché os16 non sa gestirli.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/ln.c`' [102.1.15]

VEDERE ANCHE

cp(1) [92.7], *link(2)* [93.23].

92.12 os16: login(1)

NOME

'**login**' - inizio di una sessione presso un terminale

SINTASSI

```
login
```

DESCRIZIONE

'**login**' richiede l'inserimento di un nominativo-utente e di una parola d'ordine. Questa coppia viene verificata consultando il file '`/etc/passwd`' e se coincide: vengono cambiati i permessi e la proprietà del file di dispositivo del terminale di controllo; viene cambiata la directory corrente in modo da farla coincidere con la directory personale dell'utente; viene avviata la shell, indicata sempre nel file '`/etc/passwd`' per quel tale utente, con i privilegi di questo. La shell, avviata così, va a rimpiazzare il processo di '**login**'.

Il programma '**login**' è fatto per essere avviato da '**getty**', non avendo altri utilizzi pratici.

FILE

File	Descrizione
'/etc/passwd'	Contiene l'elenco degli utenti, con l'associazione al numero UID, alla parola d'ordine necessaria per accedere, alla shell dell'utente. Le altre informazioni eventuali contenute nel file, non sono usate da '**login**'.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/login.c' [102.1.16]

VEDERE ANCHE

getty(8) [98.1], *console(4)* [95.1].

92.13 os16: ls(1)

NOME

'**ls**' - elenco del contenuto delle directory

SINTASSI

ls [*opzioni*] [*file*]...

DESCRIZIONE

'**ls**' elenca i file e le directory indicati come argomenti della chiamata. Se non vengono indicati file o directory da visualizzare, si ottiene l'elenco del contenuto della directory corrente; inoltre, questa realizzazione particolare di '**ls**', se si indica come argomento solo una directory, ne mostra il contenuto, altrimenti, se gli argomenti sono più di uno, mostra solo i nomi richiesti, eventualmente con le rispettive caratteristiche se è stata usata l'opzione '**-l**'.

OPZIONI

Opzione	Descrizione
-a	Quando si richiede di mostrare il contenuto di una directory (quella corrente o quella specificata espressamente come primo e unico argomento), con questa opzione si ottiene la visualizzazione anche dei nomi che iniziano con un punto, inclusi '.' e '..'.
-l	Con questa opzione si ottiene la visualizzazione di più informazioni sui file e sulle directory elencati.

NOTE

Dal momento che os16 non considera i gruppi, quando si usa l'opzione '**-l**', il nome del gruppo a cui appartiene un file o una directory, non viene visualizzato.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/ls.c' [102.1.17]

92.14 os16: man(1)

«

NOME

'**man**' - visualizzazione delle pagine di manuale

SINTASSI

man [***sezione***] ***pagina***

DESCRIZIONE

'**man**' visualizza la pagina di manuale indicata come argomento, consentendone lo scorrimento in avanti. La «pagina» viene cercata tra le sezioni, a partire dalla prima. In caso di omonimie tra sezioni differenti, si può specificare il numero della sezione prima del nome della pagina.

Le pagine di manuale di os16 sono semplicemente dei file di testo, collocati nella directory '/usr/share/man/', con nomi del tipo '***pagina*.*n***', dove ***n*** è il numero della sezione.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/man.c' [102.1.18]

VEDERE ANCHE

cat(1) [92.3].

92.15 os16: mkdir(1)

«

NOME

'**`mkdir`**' - creazione di directory

SINTASSI

```
mkdir [-p] [-m mod_ottale] [directory]...
```

DESCRIZIONE

'**`mkdir`**' crea una o più directory, corrispondenti ai nomi che costituiscono gli argomenti.

OPZIONI

Opzione	Descrizione
`-p`	Se la directory che si vuole creare, può richiedere prima la creazione di altre directory, con questa opzione (*parents*) si generano tutte le directory genitrici necessarie, purché quei nomi non siano già usati per dei file.
`-m` ***mod_ottale***	Quando si crea una directory, senza specificare questa opzione, si ottengono i permessi 0777_8 meno quanto contenuto nella maschera di creazione dei file e delle directory. Con l'opzione '**`-m`**' si vanno invece a specificare i permessi desiderati in modo esplicito.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/mkdir.c`' [102.1.19]

VEDERE ANCHE

mkdir(2) [93.25], *rmdir(2)* [93.30].

92.16 os16: more(1)

«

NOME

'**more**' - visualizzazione di file sullo schermo, permettendo il controllo dello scorrimento dei dati, ma in un solo verso

SINTASSI

```
more file...
```

DESCRIZIONE

'**more**' visualizza i file indicati come argomenti della chiamata, sospendendo lo scorrimento del testo dopo un certo numero di righe, consentendo all'utente di decidere come proseguire.

COMANDI

Quando '**more**' sospende lo scorrimento del testo, è possibile introdurre un comando, composto da un solo carattere, tenendo conto che ciò che non è previsto fa comunque proseguire lo scorrimento:

Comando	Descrizione
[*n*]	si richiede di saltare al file successivo, ammesso che ce ne sia un altro;
[*q*]	si richiede di interrompere lo scorrimento e di concludere il funzionamento del programma;
[*Spazio*]	si richiede di proseguire nella visualizzazione progressiva dei file.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/more.c`' [102.1.20]

VEDERE ANCHE

cat(1) [92.3].

92.17 os16: ps(1)

«

NOME

'**ps**' - visualizzazione dello stato dei processi elaborativi

SINTASSI

```
ps
```

DESCRIZIONE

'**ps**' visualizza l'elenco dei processi, con le informazioni disponibili sul loro stato. L'elenco è provvisto di un'intestazione, come si vede nell'esempio seguente:

```
pp  p pg
id id rp  tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name
 0  0  0 0000    0    0    0 00.31 r 10500 eff8 00500 0000 ffca os16 kernel
 0  1  0 0000    0    0    0 00.15 s 2f500 3200 32700 3300 2b8c /bin/init
 1  2  2 0500 1001 1001 1001 00.15 s 25700 3900 29000 3400 2f8c /bin/shell
 1  3  3 0501    0    0    0 00.07 s 22600 3100 38e00 3400 2556 /bin/login
 2  4  2 0500 1001 1001 1001 00.03 R 1f500 2b00 3c200 3400 3172 /bin/ps
```

La prima colonna, con la sigla «ppid», ovvero *parent pid*, riporta il numero del processo genitore; la seconda, con la sigla «pid», *process id*, indica il numero del processo preso in considerazione; la terza, con la sigla «pgrp», *process group*, indica il gruppo a cui appartiene il processo; la quarta colonna, «tty», indica il terminale associato, ammesso che ci sia, come numero di dispositivo, ma in base sedici. Le colonne «uid», «euid» e «suid», riguardano l'identità dell'utente, per conto del quale sono in funzione i processi, rappresentando, nell'ordine, l'identità reale (*real user id*), quella efficace (*effective user id*) e quella salvata precedentemente (*saved user id*).

La colonna «usage» indica il tempo di utilizzo della CPU; la colonna «s» indica lo stato del processo, il cui significato può essere interpretato con l'aiuto della tabella successiva:

Lettera	Significato	Descrizione
R	*running*	in corso di esecuzione
r	*ready*	pronto per essere messo in esecuzione
s	*sleeping*	in attesa
z	*zombie*	terminato e non più in memoria, per il quale si attende di passare lo stato di uscita al processo genitore.

Le colonne «iaddr» e «isiz» indicano l'indirizzo iniziale e l'estensione dell'area codice del processo, in memoria; le colonne «daddr» e «dsiz» indicano l'indirizzo iniziale e l'estensione dell'area dati del processo, in memoria. La colonna «sp» indica il valore dell'indice della pila dei dati (*stack pointer*).

L'ultima colonna indica il nome del programma, assieme al suo percorso, con il quale il processo è stato avviato.

NOTE

L'elenco dei processi include anche il kernel, il quale occupa correttamente la prima posizione (processo zero).

FILE

'**ps**' trae le informazioni sullo stato dei processi da un file di dispositivo speciale: '/dev/kmem_ps'.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/ps.c' [102.1.22]

92.18 os16: rm(1)

«

NOME

'**rm**' - cancellazione di file

SINTASSI

```
rm file...
```

DESCRIZIONE

'**rm**' consente di cancellare i file indicati come argomento.

DIFETTI

Non è possibile eseguire la cancellazione ricorsiva di una directory.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/rm.c' [102.1.23]

VEDERE ANCHE

unlink(2) [93.42].

92.19 os16: shell(1)

«

NOME

'**shell**' - interprete dei comandi

SINTASSI

```
shell
```

DESCRIZIONE

'**shell**' è l'interprete dei comandi di os16. Di norma viene avviato da '**login**', in base alla configurazione contenuta nel file '/etc/passwd'.

'**shell**' interpreta i comandi inseriti; se si tratta di un comando interno lo esegue direttamente, altrimenti cerca e avvia un programma con il nome corrispondente, rimanendo in attesa fino alla conclusione del processo relativo, per riprendere poi il controllo.

DIFETTI

L'interpretazione della riga di comando è letterale, pertanto non c'è alcuna espansione di caratteri speciali, variabili di ambiente o altro; inoltre, non è possibile eseguire script.

A volte, quando un processo avviato da '**shell**' termina di funzionare, il processo di '**shell**' non viene risvegliato correttamente, rendendo inutilizzabile il terminale. Per ovviare all'inconveniente, si può premere la combinazione [*Ctrl c*], con la quale viene inviato il segnale '**SIGINT**' a tutti i processi del gruppo associato al terminale.

Anche il fatto che un segnale generato con una combinazione di tasti si trasmetta a tutti i processi del gruppo associato al terminale è un'anomalia, tuttavia fa parte delle particolarità dovute alla semplificazione di os16.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/shell.c' [102.1.24]

VEDERE ANCHE

input_line(3) [94.60].

92.20 os16: touch(1)

NOME

'**touch**' - creazione di un file vuoto oppure aggiornamento della data di modifica

SINTASSI

```
touch file...
```

DESCRIZIONE

'**touch**' crea dei file vuoti, se quelli indicati come argomento non sono esistenti; altrimenti, aggiorna le date di accesso e di modifica, sulla base dello stato dell'orologio di sistema.

DIFETTI

Non è possibile attribuire una data arbitraria; inoltre, a causa della limitazione del tipo di file system utilizzato, non è possibile distinguere tra date di accesso e modifica dei file.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/touch.c`' [102.1.25]

92.21 os16: tty(1)

«

NOME

'**tty**' - nome del file di dispositivo del terminale associato allo standard input

SINTASSI

```
tty
```

DESCRIZIONE

Il programma '**tty**' individua il dispositivo del terminale associato allo standard input e lo traduce in un percorso che descrive il file di dispositivo corrispondente (ovvero il file di dispositivo che dovrebbe corrispondergli)

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/tty.c`' [102.1.26]

Capitolo 93

Sezione 2: chiamate di sistema

TXT
htm
PDF
pdf
P1
P2
P3
P4
epub

93.1 os16: _Exit(2)

Vedere _*exit(2)* [93.2].

93.2 os16: _exit(2)

NOME

'**_exit**', '**_Exit**' - conclusione del processo chiamante

SINTASSI

```
#include <unistd.h>
void _exit (int status);
```

```
#include <stdlib.h>
void _Exit (int status);
```

DESCRIZIONE

Le funzioni ***_exit()*** e ***_Exit()*** sono equivalenti e servono per concludere il processo chiamante, con un valore pari a quello indicato come argomento (***status***), purché inferiore o uguale 255 (FF_{16}).

La conclusione del processo implica anche la chiusura dei suoi file aperti, e l'affidamento di eventuali processi figli al processo numero uno ('**init**'); inoltre, si ottiene l'invio di un segnale SIGCHLD al processo genitore di quello che viene concluso.

VALORE RESTITUITO

La funzione non può restituire alcunché, dal momento che la sua esecuzione comporta la morte del processo.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/_exit.c`' [101.17.1]

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/_Exit.c`' [101.10.1]

'`lib/sys/os16/sys.s`' [101.12.15]

'kernel/proc/_isr.s' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

'kernel/proc/proc_sys_exit.c' [100.9.22]

VEDERE ANCHE

execve(2) [93.10], *fork(2)* [93.14], *kill(2)* [93.22], *wait(2)* [93.43], *atexit(3)* [94.4], *exit(3)* [94.4].

93.3 os16: chdir(2)

«

NOME

'**chdir**' - modifica della directory corrente

SINTASSI

```
#include <unistd.h>
int chdir (const char *path);
```

DESCRIZIONE

La funzione ***chdir()*** cambia la directory corrente, in modo che quella nuova corrisponda al percorso annotato nella stringa ***path***.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
EINVAL	Argomento non valido.
EACCES	Accesso negato.
ENOTDIR	Uno dei componenti del percorso non è una directory.
ENOENT	Uno dei componenti del percorso non esiste.

FILE SORGENTI

'lib/unistd.h' [101.17]

'lib/unistd/chdir.c' [101.17.3]

'lib/sys/os16/sys.s' [101.12.15]

'kernel/proc/_isr.s' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

'kernel/fs/path_chdir.c' [100.4.32]

VEDERE ANCHE

rmdir(2) [93.30], *access(3)* [94.1].

93.4 os16: chmod(2)

NOME

'**chmod**', '**fchmod**' - cambiamento della modalità dei permessi di un file

SINTASSI

```
#include <sys/stat.h>
int chmod  (const char *path, mode_t mode);
int fchmod (int fdn, mode_t mode);
```

DESCRIZIONE

Le funzioni ***chmod()*** e ***fchmod()*** consentono di modificare la modalità dei permessi di accesso di un file. La funzione ***chmod()*** individua il file su cui intervenire attraverso un percorso, ovvero la stringa ***path***; la funzione ***fchmod()***, invece, richiede l'indicazione del numero di un descrittore di file, già aperto. In entrambi i casi, l'ultimo argomento serve a specificare la nuova modalità dei permessi.

Tradizionalmente, i permessi si scrivono attraverso un numero in base otto; in alternativa, si possono usare convenientemente della macro-variabili, dichiarate nel file 'sys/stat.h', combinate assieme con l'operatore binario OR.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXU`	00700_8	Lettura, scrittura ed esecuzione per l'utente proprietario.
`S_IRUSR`	00400_8	Lettura per l'utente proprietario.
`S_IWUSR`	00200_8	Scrittura per l'utente proprietario.
`S_IXUSR`	00100_8	Esecuzione per l'utente proprietario.
`S_IRWXG`	00070_8	Lettura, scrittura ed esecuzione per il gruppo.
`S_IRGRP`	00040_8	Lettura per il gruppo.
`S_IWGRP`	00020_8	Scrittura per il gruppo.
`S_IXGRP`	00010_8	Esecuzione per il gruppo.
`S_IRWXO`	00007_8	Lettura, scrittura ed esecuzione per gli altri utenti.
`S_IROTH`	00004_8	Lettura per gli altri utenti.
`S_IWOTH`	00002_8	Scrittura per gli altri utenti.
`S_IXOTH`	00001_8	Esecuzione per gli altri utenti.

os16 non considera i permessi SUID (*Set user id*), SGID (*Set group id*) e Sticky, che nella tabella non sono stati nemmeno annotati; inoltre, non tiene in considerazione i permessi legati al gruppo.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EACCES`	Permesso negato.
`EBADF`	Il descrittore del file richiesto non è valido.

FILE SORGENTI

'`lib/sys/stat.h`' [101.13]

'`lib/sys/stat/chmod.c`' [101.13.1]

'`lib/sys/stat/fchmod.c`' [101.13.2]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/path_chmod.c`' [100.4.33]

VEDERE ANCHE

chmod(1) [92.5], *chown(2)* [93.5], *open(2)* [93.28], *stat(2)* [93.36].

93.5 os16: chown(2)

NOME

'**`chown`**', '**`fchown`**' - modifica della proprietà dei file

SINTASSI

```
#include <unistd.h>
int chown  (const char *path, uid_t uid, gid_t gid);
int fchown (int fdn, uid_t uid, gid_t gid);
```

DESCRIZIONE

Le funzioni ***chown()*** e ***fchown()***, modificano la proprietà di un file, fornendo il numero UID e il numero GID. Il file viene indicato, rispettivamente, attraverso il percorso scritto in una stringa, oppure come numero di descrittore già aperto.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EINVAL`	Argomento non valido.
`EPERM`	Permessi insufficienti per eseguire l'operazione.
`ENOTDIR`	Uno dei componenti del percorso non è una directory.
`ENOENT`	Uno dei componenti del percorso non esiste.
`EBADF`	Il descrittore del file non è valido.

DIFETTI

Le funzioni consentono di attribuire il numero del gruppo, ma os16 non valuta i permessi di accesso ai file, relativi ai gruppi.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/chown.c`' [101.17.4]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/path_chown.c`' [100.4.34]

'`kernel/fs/fd_chown.c`' [100.4.2]

VEDERE ANCHE

chmod(2) [93.4].

93.6 os16: clock(2)

«

NOME

'**`clock`**' - tempo della CPU espresso in unità '**`clock_t`**'

SINTASSI

```
#include <time.h>
clock_t clock (void);
```

DESCRIZIONE

La funzione ***clock()*** restituisce il tempo di utilizzo della CPU, espresso in unità '**clock_t**', corrispondenti a secondi diviso il valore della macro-variabile ***CLOCKS_PER_SEC***. Per os16, come dichiarato nel file 'time.h', il valore di ***CLOCKS_PER_SEC*** è 18, essendo la frequenza di CPU poco più di 18 Hz.

VALORE RESTITUITO

La funzione restituisce il tempo di CPU, espresso in multipli di 1/18 di secondo.

FILE SORGENTI

'lib/time.h' [101.16]

'lib/time/clock.c' [101.16.2]

'lib/sys/os16/sys.s' [101.12.15]

'kernel/proc/_isr.s' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

'kernel/k_libc/k_clock.c' [100.6.1]

VEDERE ANCHE

time(2) [93.39].

93.7 os16: close(2)

«

NOME

'**close**' - chiusura di un descrittore di file

SINTASSI

```
#include <unistd.h>
int close (int fdn);
```

DESCRIZIONE

Le funzioni ***close()*** chiude un descrittore di file.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore del file non è valido.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/close.c`' [101.17.5]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/fd_close.c`' [100.4.3]

VEDERE ANCHE

fcntl(2) [93.13], *open(2)* [93.28], *fclose(3)* [94.27].

93.8 os16: dup(2)

«

NOME

'**`dup`**', '**`dup2`**' - duplicazione di descrittori di file

SINTASSI

```
#include <unistd.h>
int dup  (int fdn_old);
int dup2 (int fdn_old, int fdn_new);
```

DESCRIZIONE

Le funzioni ***dup()*** e ***dup2()*** servono a duplicare un descrittore di file. La funzione ***dup()*** duplica il descrittore ***fdn_old***, utilizzando il numero di descrittore libero più basso che sia disponibile; ***dup2()***, invece, richiede che il nuovo numero di descrittore sia specificato, attraverso il parametro ***fdn_new***. Tuttavia, se il numero di descrittore ***fdn_new*** risulta utilizzato, questo viene chiuso prima di diventare la copia di ***fdn_old***.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EBADF`	Uno dei descrittori specificati non è valido.
`EMFILE`	Troppi file aperti per il processo.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/dup.c`' [101.17.6]

'`lib/unistd/dup2.c`' [101.17.7]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/fd_dup.c`' [100.4.4]

'`kernel/fs/fd_dup2.c`' [100.4.5]

VEDERE ANCHE

close(2) [93.7], *fcntl(2)* [93.13], *open(2)* [93.28].

93.9 os16: dup2(2)

Vedere *dup(2)* [93.8].

93.10 os16: execve(2)

NOME

'**`execve`**' - esecuzione di un file

SINTASSI

```
#include <unistd.h>
int execve (const char *path, char *const argv[],
            char *const envp[]);
```

DESCRIZIONE

La funzione ***execve()*** è quella che avvia effettivamente un programma, mentre le altre funzioni '**`exec...()`**' offrono semplicemente un'interfaccia differente per

l'avvio, ma poi si avvalgono di ***execve()*** per svolgere effettivamente quanto loro richiesto.

La funzione ***execve()*** avvia il file il cui percorso è specificato come stringa, nel primo argomento.

Il secondo argomento deve essere un array di stringhe, dove la prima deve rappresentare il nome del programma avviato e le successive sono gli argomenti da passare al programma. L'ultimo elemento di tale array deve essere un puntatore nullo, per poter essere riconosciuto.

Il terzo argomento deve essere un array di stringhe, rappresentanti l'ambiente da passare al nuovo processo. Per ambiente si intende l'insieme delle variabili di ambiente, pertanto queste stringhe devono avere la forma '***nome=valore***', per essere riconoscibili. Anche in questo caso, per poter individuare l'ultimo elemento dell'array, questo deve essere un puntatore nullo.

VALORE RESTITUITO

Se ***execve()*** riesce nel suo compito, non può restituire alcunché, dato che in quel momento, il processo chiamante viene rimpiazzato da quello del file che viene eseguito. Pertanto, se viene restituito qualcosa, può trattarsi solo di un valore che rappresenta un errore, ovvero -1, aggiornando anche la variabile ***errno*** di conseguenza.

ERRORI

Valore di ***errno***	Significato
`E2BIG`	Ci sono troppi argomenti.
`ENOMEM`	Memoria insufficiente.
`ENOENT`	Il file richiesto non esiste.
`EACCES`	Il file non può essere avviato per la mancanza dei permessi di acceso necessari.
`ENOEXEC`	Il file non può essere un file eseguibile, perché non ne ha le caratteristiche.
`EIO`	Errore di input-output.

DIFETTI

os16 non prevede l'interpretazione di script, perché non esiste alcun programma in grado di farlo. Anche la shell di os16 si limita a eseguire i comandi inseriti, ma non può interpretare un file.

FILE SORGENTI

'lib/unistd.h' [101.17]

'lib/unistd/execle.c' [101.17.10]

'lib/sys/os16/sys.s' [101.12.15]

'kernel/proc/_isr.s' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

'kernel/proc/proc_sys_exec.c' [100.9.21]

VEDERE ANCHE

fork(2) [93.14], *exec(3)* [94.20], *getopt(3)* [94.52], *environ(7)* [97.1].

93.11 os16: fchmod(2)

Vedere *chmod(2)* [93.4].

93.12 os16: fchown(2)

Vedere *chown(2)* [93.5].

93.13 os16: fcntl(2)

NOME

'**fcntl**' - configurazione e intervento sui descrittori di file

SINTASSI

```
#include <fcntl.h>
int fcntl (int fdn, int cmd, ...);
```

DESCRIZIONE

La funzione ***fcntl()*** esegue un'operazione, definita dal parametro ***cmd***, sul descrittore richiesto come primo parametro (***fdn***). A seconda del tipo di operazione richiesta, potrebbero essere necessari degli argomenti ulteriori, i quali però non possono essere formalizzati in modo esatto nel prototipo della funzione. Il valore del secondo parametro che rappresenta l'operazione richiesta, va fornito in forma di costante simbolica, come descritto nell'elenco seguente.

Sintassi	Descrizione
fcntl (*fdn*, F_DUPFD, (int) *fdn_min*)	Richiede la duplicazione del descrittore di file *fdn*, in modo tale che la copia abbia il numero di descrittore minore possibile, ma maggiore o uguale a quello indicato come argomento *fdn_min*.
fcntl (*fdn*, F_GETFD) fcntl (*fdn*, F_SETFD, (int) *fd_flags*)	Rispettivamente, legge o imposta, gli indicatori del descrittore di file *fdn* (eventualmente noti come *file descriptor flags* o solo *fd flags*). Per il momento, è possibile impostare un solo indicatore, '**FD_CLOEXEC**', pertanto, al posto di *fd_flags* si può mettere solo la costante '**FD_CLOEXEC**'.
fcntl (*fdn*, F_GETFL) fcntl (*fdn*, F_SETFL, (int) *fl_flags*)	Rispettivamente, legge o imposta, gli indicatori dello stato del file, relativi al descrittore *fdn* (eventualmente noti come *file flaga* o solo *fl flags*). Per impostare questi indicatori, vanno combinate delle costanti simboliche: '**O_RDONLY**', '**O_WRONLY**', '**O_RDWR**', '**O_CREAT**', '**O_EXCL**', '**O_NOCTTY**', '**O_TRUNC**'.

VALORE RESTITUITO

Il significato del valore restituito dalla funzione dipende dal tipo di operazione richiesta, come sintetizzato dalla tabella successiva.

Operazione richiesta	Significato del valore restituito
`F_DUPFD`	Si ottiene il numero del descrittore prodotto dalla copia, oppure −1 in caso di errore.
`F_GETFD`	Si ottiene il valore degli indicatori del descrittore (*fd flags*), oppure −1 in caso di errore.
`F_GETFL`	Si ottiene il valore degli indicatori del file (*fl flags*), oppure −1 in caso di errore.

`F_GETOWN`
`F_SETOWN`
`F_GETLK`
`F_SETLK`
`F_SETLKW`
	Si ottiene −1, in quanto si tratta di operazioni non realizzate in questa versione della funzione, per os16.

altri tipi di operazione	Si ottiene 0 in caso di successo, oppure −1 in caso di errore.

ERRORI

Valore di ***errno***	Significato
`E_NOT_IMPLEMENTED`	È stato richiesto un tipo di operazione che non è disponibile nel caso particolare di os16.
`EINVAL`	È stato richiesto un tipo di operazione non valido.

FILE SORGENTI

'`lib/fcntl.h`' [101.4]

'`lib/fcntl/fcntl.c`' [101.4.2]

VEDERE ANCHE

dup(2) [93.8], *dup2(2)* [93.8], *open(2)* [93.28].

93.14 os16: fork(2)

«

NOME

'**fork**' - sdoppiamento di un processo, ovvero creazione di un processo figlio

SINTASSI

```
#include <unistd.h>
pid_t fork (void);
```

DESCRIZIONE

La funzione ***fork()*** crea una copia del processo in corso, la quale copia diventa un processo figlio del primo. Il processo figlio eredita una copia dei descrittori di file aperti e di conseguenza dei flussi di file e directory.

Il processo genitore riceve dalla funzione il valore del numero PID del processo figlio avviato; il processo figlio si mette a funzionare dal punto in cui si trova la funzione ***fork()***, restituendo però un valore nullo: in questo modo tale processo figlio può riconoscersi come tale.

VALORE RESTITUITO

La funzione restituisce al processo genitore il numero PID del processo figlio; al processo figlio restituisce zero. In caso di problemi, invece, il valore restituito è -1 e la variabile ***errno*** risulta aggiornata di conseguenza.

ERRORI

Valore di ***errno***	Significato
`ENOMEM`	Memoria insufficiente per avviare un altro processo.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/fork.c`' [101.17.17]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_fork.c`' [100.9.23]

VEDERE ANCHE

execve(2) [93.10], *wait(2)* [93.43], *exec(3)* [94.20].

93.15 os16: fstat(2)

Vedere *stat(2)* [93.36].

93.16 os16: getcwd(2)

NOME

'`getcwd`' - determinazione della directory corrente

SINTASSI

```
#include <unistd.h>
char *getcwd (char *buffer, size_t size);
```

DESCRIZIONE

La funzione ***getcwd()*** modifica il contenuto dell'area di memoria a cui punta ***buffer***, copiandovi al suo interno la stringa che rappresenta il percorso della directory corrente. La scrittura all'interno di ***buffer*** può prolungarsi al massimo per ***size*** byte, incluso il codice nullo di terminazione delle stringhe.

VALORE RESTITUITO

La funzione restituisce il puntatore alla stringa che rappresenta il percorso della directory corrente, il quale deve coincidere con ***buffer***. In caso di errore, invece, la funzione restituisce il puntatore nullo '**`NULL`**'.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il puntatore ***buffer*** non è valido.
`E_LIMIT`	Il percorso della directory corrente è troppo lungo, rispetto ai limiti realizzativi di os16.

DIFETTI

La funzione ***getcwd()*** di os16 deve comunicare con il kernel per ottenere l'informazione che le serve, perché la «u-area» (*User area*) è trattenuta all'interno del kernel stesso.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/getcwd.c`' [101.17.18]

‘`lib/sys/os16/sys.s`’ [101.12.15]

‘`kernel/proc/_isr.s`’ [100.9.1]

‘`kernel/proc/sysroutine.c`’ [100.9.30]

VEDERE ANCHE

chdir(2) [93.3].

93.17 os16: geteuid(2)

«

Vedere *getuid(2)* [93.18].

93.18 os16: getuid(2)

«

NOME

‘**`getuid`**’, ‘**`geteuid`**’ - determinazione dell’identità reale ed efficace

SINTASSI

```
#include <unistd.h>
uid_t getuid  (void);
uid_t geteuid (void);
```

DESCRIZIONE

La funzione ***getuid()*** restituisce il numero corrispondente all’identità reale del processo; la funzione ***geteuid()*** restituisce il numero dell’identità efficace del processo.

VALORE RESTITUITO

Il numero UID, reale o efficace del processo chiamante. Non sono previsti casi di errore.

DIFETTI

Le funzioni ***getuid()*** e ***geteuid()*** di os16 devono comunicare con il kernel per ottenere l’informazione che a loro serve, perché la «u-area» (*User area*) è trattenuta all’interno del kernel stesso.

FILE SORGENTI

‘`lib/unistd.h`’ [101.17]

‘`lib/unistd/getuid.c`’ [101.17.24]

‘`lib/unistd/geteuid.c`’ [101.17.19]

‘`lib/sys/os16/sys.s`’ [101.12.15]

‘`kernel/proc/_isr.s`’ [100.9.1]

‘`kernel/proc/sysroutine.c`’ [100.9.30]

VEDERE ANCHE

setuid(2) [93.33].

93.19 os16: getpgrp(2)

Vedere *getpid(2)* [93.20].

93.20 os16: getpid(2)

NOME

‘**`getpid`**’, ‘**`getppid`**’, ‘**`getpgrp`**’ - determinazione del numero del processo o del gruppo di processi

SINTASSI

```
#include <unistd.h>
pid_t getpid  (void);
pid_t getppid (void);
pid_t getpgrp (void);
```

DESCRIZIONE

La funzione ***getpid()*** restituisce il numero del processo chiamante; la funzione ***getppid()*** restituisce il numero del processo genitore rispetto a quello chiamante; la funzione ***getpgrp()*** restituisce il numero attribuito al gruppo di processi a cui appartiene quello chiamante.

VALORE RESTITUITO

Il numero di processo o di gruppo di processi, relativo al contesto della funzione. Non sono previsti casi di errore.

DIFETTI

Le funzioni ***getpid()***, ***getppid()*** e ***getpgrp()*** di os16 devono comunicare con il kernel per ottenere l’informazione che a loro serve, perché la «u-area» (*User area*) è trattenuta all’interno del kernel stesso.

FILE SORGENTI

‘`lib/unistd.h`’ [101.17]

‘`lib/unistd/getpid.c`’ [101.17.22]

‘`lib/unistd/getppid.c`’ [101.17.23]

‘`lib/unistd/getpgrp.c`’ [101.17.21]

‘`lib/sys/os16/sys.s`’ [101.12.15]

‘`kernel/proc/_isr.s`’ [100.9.1]

‘`kernel/proc/sysroutine.c`’ [100.9.30]

VEDERE ANCHE

getuid(2) [93.18] *fork(2)* [93.14], *execve(2)* [93.10].

93.21 os16: getppid(2)

Vedere *getpid(2)* [93.20].

93.22 os16: kill(2)

NOME

‘**`kill`**’ - invio di un segnale a un processo

SINTASSI

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig)
```

DESCRIZIONE

La funzione ***kill()*** invia il segnale ***sig*** al processo numero ***pid***, oppure a un gruppo di processi. Questa realizzazione particolare di os16 comporta come segue:

- se il valore ***pid*** è maggiore di zero, il segnale viene inviato al processo con il numero ***pid***, ammesso di averne il permesso;
- se il valore ***pid*** è pari a zero, il segnale viene inviato a tutti i processi appartenenti allo stesso utente (quelli che hanno la stessa identità efficace, ovvero il valore ***euid***), ma se il processo che chiama la funzione lavora con un valore di ***euid*** pari a zero, il segnale viene inviato a tutti i processi, a partire dal numero due (si salta ‘**`init`**’);

- valori negativi di ***pid*** non vengono presi in considerazione.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EPERM`	Il processo non ha i permessi per inviare il segnale alla destinazione richiesta.
`ESRCH`	La ricerca del processo ***pid*** è fallita. Nel caso di os16, si ottiene questo errore anche per valori negativi di ***pid***.

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/signal.h`' [101.8]

'`lib/signal/kill.c`' [101.8.1]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_kill.c`' [100.9.24]

VEDERE ANCHE

signal(2) [93.34].

93.23 os16: link(2)

NOME

'**`link`**' - creazione di un collegamento fisico tra un file esistente e un altro nome

SINTASSI

```
#include <unistd.h>
int link (const char *path_old, const char *path_new);
```

DESCRIZIONE

La funzione ***link()*** produce un nuovo collegamento a un file già esistente. Va fornito il percorso del file già esistente, ***path_old*** e quello del file da creare, in

qualità di collegamento, ***path_new***. L'operazione può avvenire soltanto se i due percorsi si trovano sulla stessa unità di memorizzazione e se ci sono i permessi di scrittura necessari nella directory di destinazione. Dopo l'operazione di collegamento, fatta in questo modo, non è possibile distinguere quale sia il file originale e quale sia invece il nome aggiunto.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
`EPERM`	Operazione non consentita.
`EEXIST`	Il nome da creare esiste già.
`EACCES`	Accesso non consentito.
`ENOENT`	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
`EROFS`	Il file system consente soltanto un accesso in lettura.
`ENOTDIR`	Uno dei due percorsi non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/link.c`' [101.17.26]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/path_link.c`' [100.4.40]

VEDERE ANCHE

ln(1) [92.11] *open(2)* [93.28], *stat(2)* [93.36], *unlink(2)* [93.42].

93.24 os16: lseek(2)

NOME

'**lseek**' - riposizionamento dell'indice di accesso a un descrittore di file

SINTASSI

```
#include <unistd.h>
off_t lseek (int fdn, off_t offset, int whence);
```

DESCRIZIONE

La funzione ***lseek()*** consente di riposizionare l'indice di accesso interno al descrittore di file ***fdn***. Per fare questo occorre prima determinare un punto di riferimento, rappresentato dal parametro ***whence***, dove va usata una macro-variabile definita nel file '`unistd.h`'. Può trattarsi dei casi seguenti.

Valore di ***whence***	Significato
`SEEK_SET`	Lo scostamento si riferisce all'inizio del file.
`SEEK_CUR`	Lo scostamento si riferisce alla posizione che ha già l'indice interno al file.
`SEEK_END`	Lo scostamento si riferisce alla fine del file.

Lo scostamento indicato dal parametro ***offset*** si applica a partire dalla posizione a cui si riferisce ***whence***, pertanto può avere segno positivo o negativo, ma in ogni caso non è possibile collocare l'indice prima dell'inizio del file.

VALORE RESTITUITO

Se l'operazione avviene con successo, la funzione restituisce il valore dell'indice riposizionato, preso come scostamento a partire dall'inizio del file. In caso di errore, restituisce invece il valore -1, aggiornando di conseguenza anche la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il numero del descrittore di file non è valido.
`EINVAL`	Il valore di ***whence*** non è contemplato, oppure la combinazione tra ***whence*** e ***offset*** non è valida.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/lseek.c`' [101.17.27]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/fd_lseek.c`' [100.4.7]

VEDERE ANCHE

dup(2) [93.8] *fork(2)* [93.14], *open(2)* [93.28], *fseek(3)* [94.43].

93.25 os16: mkdir(2)

«

NOME

'**`mkdir`**' - creazione di una directory

SINTASSI

```
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode);
```

DESCRIZIONE

La funzione ***mkdir()*** crea una directory, indicata attraverso un percorso, nel parametro ***path***, specificando la modalità dei permessi, con il parametro ***mode***.

Tuttavia, il valore del parametro ***mode*** non viene preso in considerazione integralmente: di questo si considerano solo gli ultimi nove bit, ovvero quelli dei permessi di utenti, gruppi e altri utenti; inoltre, vengono tolti i bit presenti nella maschera dei permessi associata al processo (si veda anche *umask(2)* [93.40]).

La directory che viene creata in questo modo, appartiene all'identità efficace del processo, ovvero all'utente per conto del quale questo sta funzionando.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EINVAL`	Il percorso indicato non è valido.
`EEXIST`	Esiste già un file o una directory con lo stesso nome.
`ENOTDIR`	Una porzione del percorso della directory da creare, non è una directory.
`ENOENT`	Una porzione del percorso della directory da creare non esiste.
`EACCES`	Permesso negato.

FILE SORGENTI

'`lib/sys/stat.h`' [101.13]

'`lib/sys/stat/mkdir.c`' [101.13.4]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/path_mkdir.c`' [100.4.41]

VEDERE ANCHE

mkdir(1) [92.15], *chmod(2)* [93.4], *chown(2)* [93.5], *mknod(2)* [93.26], *mount(2)* [93.27], *stat(2)* [93.36], *umask(2)* [93.40], *unlink(2)* [93.42].

93.26 os16: mknod(2)

NOME

'**`mknod`**' - creazione di un file vuoto di qualunque tipo

SINTASSI

```
#include <sys/stat.h>
int mknod (const char *path, mode_t mode, dev_t device);
```

DESCRIZIONE

La funzione ***mknod()*** crea un file vuoto, di qualunque tipo. Potenzialmente può creare anche una directory, ma priva di qualunque voce, rendendola così non

adeguata al suo scopo (una directory richiede almeno le voci '`.`' e '`..`', per potersi considerare tale).

Il parametro ***path*** specifica il percorso del file da creare; il parametro ***mode*** serve a indicare il tipo di file da creare, oltre ai permessi comuni.

Il parametro ***device***, con il quale va indicato il numero di un dispositivo (completo di numero primario e secondario), viene preso in considerazione soltanto se nel parametro ***mode*** si richiede la creazione di un file di dispositivo a caratteri o a blocchi.

Il valore del parametro ***mode*** va costruito combinando assieme delle macro-variabili definite nel file '`sys/stat.h`', come descritto nella pagina di manuale *stat(2)* [93.36], tenendo conto che os16 non può gestire file FIFO, collegamenti simbolici e socket di dominio Unix.

Il valore del parametro ***mode***, per la porzione che riguarda i permessi di accesso al file, viene comunque filtrato con la maschera dei permessi (*umask(2)* [93.36]).

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il percorso indicato non è valido.
`EEXIST`	Esiste già un file o una directory con lo stesso nome.
`ENOTDIR`	Una porzione del percorso del file da creare, non è una directory.
`ENOENT`	Una porzione del percorso del file da creare non esiste.
`EACCES`	Permesso negato.

FILE SORGENTI

'`lib/sys/stat.h`' [101.13]

'`lib/sys/stat/mknod.c`' [101.13.5]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'kernel/fs/path_mknod.c' [100.4.42]

VEDERE ANCHE

mkdir(2) [93.25], *chmod(2)* [93.4], *chown(2)* [93.5], *fcntl(2)* [93.13], *stat(2)* [93.36], *umask(2)* [93.40], *unlink(2)* [93.42].

93.27 os16: mount(2)

NOME

'**mount**', '**umount**' - innesto e distacco di unità di memorizzazione

SINTASSI

```
#include <sys/os16.h>
int mount  (const char *path_dev, const char *path_mnt,
            int options);
int umount (const char *path_mnt);
```

DESCRIZIONE

La funzione ***mount()*** permette l'innesto di un'unità di memorizzazione individuata attraverso il percorso del file di dispositivo nel parametro ***path_dev***, nella directory corrispondente al percorso ***path_mnt***, con le opzioni indicate numericamente nell'ultimo argomento ***options***. Le opzioni di innesto, rappresentate attraverso delle macro-variabili, sono solo due:

Opzione	Descrizione
`MOUNT_DEFAULT`	Innesto normale, in lettura e scrittura.
`MOUNT_RO`	Innesto in sola lettura.

La funzione ***umount()*** consente di staccare un innesto fatto precedentemente, specificando il percorso della directory in cui questo è avvenuto.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Errore: va verificato il contenuto della variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EPERM`	Problema di accesso dovuto alla mancanza dei permessi necessari.
`ENOTDIR`	Ciò che dovrebbe essere una directory, non lo è.
`EBUSY`	La directory innesta già un file system e non può innestarne un altro.
`ENOENT`	La directory non esiste.
`E_NOT_MOUNTED`	La directory non innesta un file system da staccare.
`EUNKNOWN`	Si è verificato un problema non previsto e sconosciuto.

FILE SORGENTI

'`lib/sys/os16.h`' [101.12]

'`lib/sys/os16/mount.c`' [101.12.12]

'`lib/sys/os16/umount.c`' [101.12.16]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/path_mount.c`' [100.4.43]

'`kernel/fs/path_umount.c`' [100.4.45]

VEDERE ANCHE

mount(8) [98.4], *umount(8)* [98.4].

93.28 os16: open(2)

«

NOME

'**`open`**' - apertura di un file puro e semplice oppure di un file di dispositivo

SINTASSI

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *path, int oflags);
int open (const char *path, int oflags, mode_t mode);
```

DESCRIZIONE

La funzione ***open()*** apre un file, indicato attraverso il percorso ***path***, in base alle opzioni rappresentate dagli indicatori ***oflags***. A seconda del tipo di indicatori specificati, potrebbe essere richiesto il parametro ***mode***.

Quando la funzione porta a termine correttamente il proprio compito, restituisce il numero del descrittore del file associato, il quale è sempre quello di valore più basso disponibile per il processo elaborativo in corso.

Il descrittore di file ottenuto inizialmente con la funzione ***open()***, è legato al processo elaborativo in corso; tuttavia, se successivamente il processo si sdoppia attraverso la funzione ***fork()***, tale descrittore, se ancora aperto, viene duplicato nella nuova copia del processo. Inoltre, se per il descrittore aperto non viene impostato l'indicatore '**`FD_CLOEXEC`**' (con l'ausilio della funzione ***fcntl()***), se il processo viene rimpiazzato con la funzione ***execve()***, il descrittore aperto viene ereditato dal nuovo processo.

Il parametro ***oflags*** richiede necessariamente la specificazione della modalità di accesso, attraverso la combinazione appropriata dei valori: '**`O_RDONLY`**', '**`O_WRONLY`**', '**`O_RDWR`**'. Inoltre, si possono combinare altri indicatori: '**`O_CREAT`**', '**`O_TRUNC`**', '**`O_APPEND`**'.

Opzione	Descrizione
`O_RDONLY`	Richiede un accesso in lettura.
`O_WRONLY`	Richiede un accesso in scrittura.

`O_RDWR`
 	
`O_RDONLY|O_WRONLY`
	Richiede un accesso in lettura e scrittura (la combinazione di '**`R_RDONLY`**' e di '**`O_WRONLY`**' è equivalente all'uso di '**`O_RDWR`**').

`O_CREAT`	Richiede di creare contestualmente il file, ma in tal caso va usato anche il parametro ***mode***.
`O_TRUNC`	Se file da aprire esiste già, richiede che questo sia ridotto preventivamente a un file vuoto.
`O_APPEND`	Fa in modo che le operazioni di scrittura avvengano sempre partendo dalla fine del file.

Quando si utilizza l'opzione '**`O_CREAT`**', è necessario stabilire la modalità dei permessi, cosa che va fatta preferibilmente attraverso la combinazione di costanti simboliche appropriate, come elencato nella tabella successiva. Tale combinazione va fatta con l'uso dell'operatore OR binario; per esempio: '**`S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH`**'. Va osservato che os16 non gestisce i gruppi di utenti, pertanto, la definizione dei permessi relativi agli utenti appartenenti al gruppo proprietario di un file, non ha poi effetti pratici nel controllo degli accessi per tale tipo di contesto.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXU`	00700_8	Lettura, scrittura ed esecuzione per l'utente proprietario.
`S_IRUSR`	00400_8	Lettura per l'utente proprietario.
`S_IWUSR`	00200_8	Scrittura per l'utente proprietario.
`S_IXUSR`	00100_8	Esecuzione per l'utente proprietario.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXG`	00070_8	Lettura, scrittura ed esecuzione per il gruppo.
`S_IRGRP`	00040_8	Lettura per il gruppo.
`S_IWGRP`	00020_8	Scrittura per il gruppo.
`S_IXGRP`	00010_8	Esecuzione per il gruppo.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXO`	00007_8	Lettura, scrittura ed esecuzione per gli altri utenti.
`S_IROTH`	00004_8	Lettura per gli altri utenti.
`S_IWOTH`	00002_8	Scrittura per gli altri utenti.
`S_IXOTH`	00001_8	Esecuzione per gli altri utenti.

VALORE RESTITUITO

La funzione restituisce il numero del descrittore del file aperto, se l'operazione ha avuto successo, altrimenti dà semplicemente -1, impostando di conseguenza il valore della variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EINVAL`	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
`EPERM`	Operazione non consentita.
`EEXIST`	Il file da creare esiste già.
`EACCES`	Accesso non consentito.
`ENOENT`	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
`EROFS`	Avendo richiesto un accesso in scrittura, si ottiene che il file system che lo contiene consente soltanto un accesso in lettura.
`ENOTDIR`	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
`ENFILE`	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
`EMFILE`	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/sys/stat.h`' [101.13]

'`lib/fcntl.h`' [101.4]

'`lib/fcntl/open.c`' [101.4.3]

VEDERE ANCHE

chmod(2) [93.4], *chown(2)* [93.5], *close(2)* [93.7], *dup(2)* [93.8], *fcntl(2)* [93.13], *link(2)* [93.24], *mknod(2)* [93.26], *mount(2)* [93.27], *read(2)* [93.29], *stat(2)* [93.36], *umask(2)* [93.40], *unlink(2)* [93.42], *write(2)* [93.44], *fopen(3)* [94.35].

93.29 os16: read(2)

«

NOME

'**`read`**' - lettura di descrittore di file

SINTASSI

```
#include <unistd.h>
ssize_t read (int fdn, void *buffer, size_t count);
```

DESCRIZIONE

La funzione ***read()*** cerca di leggere il file rappresentato dal descrittore ***fdn***, partendo dalla posizione in cui si trova l'indice interno di accesso, per un massimo di ***count*** byte, collocando i dati letti in memoria a partire dal puntatore ***buffer***. L'indice interno al file viene fatto avanzare della quantità di byte letti effettivamente, se invece si incontra la fine del file, viene aggiornato l'indicatore interno per segnalare tale fatto.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letti effettivamente, oppure zero se è stata raggiunta la fine del file e non si può proseguire oltre. Va osservato che la lettura effettiva di una quantità inferiore di byte rispetto a quanto richiesto non costituisce un errore: in quel caso i byte mancanti vanno richiesti con successive operazioni di lettura. In caso di errore, la funzione restituisce il valore -1, aggiornando contestualmente la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il numero del descrittore di file non è valido.
`EINVAL`	Il file non è aperto in lettura.
`E_FILE_TYPE_UNSUPPORTED`	Il file è di tipo non gestibile con os16.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/read.c`' [101.17.28]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/fs/fd_read.c`' [100.4.9]

VEDERE ANCHE

close(2) [93.7] *open(2)* [93.28], *write(2)* [93.44].

93.30 os16: rmdir(2)

NOME

'**rmdir**' - eliminazione di una directory vuota

SINTASSI

```
#include <unistd.h>
int rmdir (const char *path);
```

DESCRIZIONE

La funzione ***rmdir()*** cancella la directory indicata come percorso, nella stringa ***path***, purché sia vuota.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il percorso ***path*** non è valido o è semplicemente un puntatore nullo.
`ENOTDIR`	Il nome indicato o le posizioni intermedie del percorso si riferiscono a qualcosa che non è una directory.
`ENOTEMPTY`	La directory che si vorrebbe cancellare non è vuota.
`EROFS`	La directory si trova in un'unità innestata in sola lettura.
`EPERM`	Mancano i permessi necessari per eseguire l'operazione.
`EUNKNOWN`	Si è verificato un errore imprevisto e sconosciuto.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/rmdir.c`' [101.17.29]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/fs/path_unlink.c`' [100.4.46]

VEDERE ANCHE

mkdir(2) [93.25], *unlink(2)* [93.42].

93.31 os16: seteuid(2)

Vedere *setuid(2)* [93.33].

93.32 os16: setpgrp(2)

NOME

'**`setpgrp`**' - impostazione del gruppo a cui appartiene il processo

SINTASSI

```
#include <unistd.h>
int setpgrp (void);
```

DESCRIZIONE

La funzione ***setpgrp()*** fa sì che il processo in corso costituisca un proprio gruppo autonomo, corrispondente al proprio numero PID. In altri termini, la funzione serve per iniziare un nuovo gruppo di processi, a cui i processi figli creati successivamente vengano associati in modo predefinito.

VALORE RESTITUITO

La funzione termina sempre con successo e restituisce sempre zero.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/setpgrp.c`' [101.17.31]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

VEDERE ANCHE

getpgrp(2) [93.32], *getuid(2)* [93.33].

93.33 os16: setuid(2)

«

NOME

'**setuid**', '**seteuid**' - impostazione dell'identità dell'utente

SINTASSI

```
#include <unistd.h>
int setuid  (uid_t uid);
int seteuid (uid_t uid);
```

DESCRIZIONE

A ogni processo viene attribuita l'identità di un utente, rappresentata da un numero, noto come UID, ovvero *user identity*. Tuttavia si distinguono tre tipi di numeri UID: l'identità reale, l'identità efficace e un'identità salvata in precedenza. L'identità efficace (EUID) è quella con cui opera sostanzialmente il processo; l'identità salvata è quella che ha avuto il processo in un altro momento in qualità di identità efficace e che per qualche motivo non ha più.

La funzione ***setuid()*** riceve come argomento un numero UID e si comporta diversamente a seconda della personalità del processo, come descritto nell'elenco successivo:

- se l'identità efficace del processo corrisponde a zero, trattandosi di un caso particolarmente privilegiato, tutte le identità del processo (reale, efficace e salvata) vengono inizializzate con il valore fornito alla funzione ***setuid()***;
- se l'identità efficace del processo corrisponde a quella fornita come argomento a ***setuid()***, nulla cambia nella gestione delle identità del processo;
- se l'identità reale o quella salvata in precedenza corrispondono a quella fornita come argomento di ***setuid()***, viene aggiornato il valore dell'identità efficace, senza cambiare le altre;
- in tutti gli altri casi, l'operazione non è consentita e si ottiene un errore.

La funzione ***seteuid()*** riceve come argomento un numero UID e imposta con tale valore l'identità efficace del processo, purché si verifichi almeno una delle condizioni seguenti:

- l'identità efficace del processo è zero;
- l'identità reale o quella salvata del processo corrisponde all'identità efficace che si vuole impostare;

- l'identità efficace del processo corrisponde già all'identità efficace che si vuole impostare.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EPERM`	Non si dispone dei permessi necessari a eseguire il cambiamento di identità.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/setuid.c`' [101.17.32]

'`lib/unistd/seteuid.c`' [101.17.30]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

VEDERE ANCHE

getuid(2) [93.33].

93.34 os16: signal(2)

«

NOME

'**`signal`**' - abilitazione e disabilitazione dei segnali

SINTASSI

```
#include <signal.h>
sighandler_t signal (int sig, sighandler_t handler);
```

DESCRIZIONE

La funzione ***signal()*** di os16 consente soltanto di abilitare o disabilitare un segnale, il quale, se abilitato, può essere gestito solo in modo predefinito dal sistema. Pertanto, il valore che può assumere ***handler*** sono solo: '**`SIG_DFL`**' (gestione

predefinita) e '**`SIG_IGN`**' (ignora il segnale). Il primo parametro, ***sig***, rappresenta il segnale a cui applicare ***handler***.

Il tipo '**`sighandler_t`**' è definito nel file '`signal.h`', nel modo seguente:

```
typedef void (*sighandler_t) (int);
```

Rappresenta il puntatore a una funzione avente un solo parametro di tipo '**`int`**', la quale non restituisce alcunché.

VALORE RESTITUITO

La funzione restituisce il tipo di «gestione» impostata precedentemente per il segnale richiesto, ovvero '**`SIG_ERR`**' in caso di errore.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il numero ***sig*** o il valore di ***handler*** non sono validi.

NOTE

Lo scopo della funzione ***signal()*** dovrebbe essere quello di consentire l'associazione di un evento, manifestato da un segnale, all'esecuzione di un'altra funzione, avente una forma del tipo '***nome* `(int)`**'. os16 non consente tale gestione, pertanto lascia soltanto la possibilità di attribuire un comportamento predefinito al segnale scelto, oppure di disabilitarlo, ammesso che ciò sia consentito. Sotto questo aspetto, il fatto di dover gestire i valori '**`SIG_ERR`**', '**`SIG_DFL`**' e '**`SIG_IGN`**', come se fossero puntatori a una funzione, diventa superfluo, ma rimane utile soltanto per mantenere un minimo di conformità con quello che è lo standard della funzione ***signal()***.

FILE SORGENTI

'`lib/signal.h`' [101.8]

'`lib/signal/signal.c`' [101.8.2]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_signal.c`' [100.9.27]

VEDERE ANCHE

kill(2) [93.22].

93.35 os16: sleep(2)

NOME

'**sleep**' - pausa volontaria del processo chiamante

SINTASSI

```
#include <unistd.h>
unsigned int sleep (unsigned int seconds);
```

DESCRIZIONE

La funzione ***sleep()*** chiede di mettere a riposo il processo chiamante per la quantità di secondi indicata come argomento. Il processo può però essere risvegliato prima della conclusione di tale durata, ma in tal caso la funzione restituisce la quantità di secondi che non sono stati usati per il «riposo» del processo.

VALORE RESTITUITO

La funzione restituisce zero se la pausa richiesta è trascorsa completamente; altrimenti, restituisce quanti secondi mancano ancora per completare il tempo di riposo chiesto originariamente. Non si prevede il manifestarsi di errori.

FILE SORGENTI

'lib/unistd.h' [101.17]

'lib/unistd/sleep.c' [101.17.33]

'lib/sys/os16/sys.s' [101.12.15]

'kernel/proc/_isr.s' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

VEDERE ANCHE

signal(2) [93.34].

93.36 os16: stat(2)

NOME

'**stat**', '**fstat**' - interrogazione dello stato di un file

SINTASSI

```
#include <sys/stat.h>
int stat  (const char *path, struct stat *buffer);
int fstat (int fdn, struct stat *buffer);
```

DESCRIZIONE

Le funzioni ***stat()*** e ***fstat()*** interrogano il sistema su di un file, per ottenerne le caratteristiche in forma di variabile strutturata di tipo '**`struct stat`**'.

La funzione ***stat()*** individua il file attraverso una stringa contenente il suo percorso (***path***); la funzione '**`fstat`**' si riferisce a un file aperto di cui si conosce il numero del descrittore (***fdn***). In entrambi i casi, la struttura che deve accogliere l'esito dell'interrogazione, viene indicata attraverso un puntatore, come ultimo argomento (***buffer***).

La struttura '**`struct stat`**' è definita nel file '`sys/stat.h`' nel modo seguente:

```
struct stat {
  dev_t     st_dev;     // Device containing the file.
  ino_t     st_ino;     // File serial number (inode
                        // number).
  mode_t    st_mode;    // File type and permissions.
  nlink_t   st_nlink;   // Links to the file.
  uid_t     st_uid;     // Owner user id.
  gid_t     st_gid;     // Owner group id.
  dev_t     st_rdev;    // Device number if it is a
                        // device file.
  off_t     st_size;    // File size.
  time_t    st_atime;   // Last access time.
  time_t    st_mtime;   // Last modification time.
  time_t    st_ctime;   // Last inode modification.
  blksize_t st_blksize; // Block size for I/O operations.
  blkcnt_t  st_blocks;  // File size / block size.
};
```

Va osservato che il file system Minix 1, usato da os16, riporta esclusivamente la data e l'ora di modifica, pertanto le altre due date previste sono sempre uguali a quella di modifica.

Il membro ***st_mode***, oltre alla modalità dei permessi che si cambiano con *chmod(2)* [93.4], serve ad annotare altre informazioni. Nel file '`sys/stat.h`'

sono definite delle macro-istruzioni, utili per individuare il tipo di file. Queste macro-istruzioni si risolvono in un valore numerico diverso da zero, solo se la condizione che rappresentano è vera:

Macro-istruzione	Significato
`S_ISBLK (`***m***`)`	È un file di dispositivo a blocchi?
`S_ISCHR (`***m***`)`	È un file di dispositivo a caratteri?
`S_ISFIFO (`***m***`)`	È un file FIFO?
`S_ISREG (`***m***`)`	È un file puro e semplice?
`S_ISDIR (`***m***`)`	È una directory?
`S_ISLNK (`***m***`)`	È un collegamento simbolico?
`S_ISSOCK (`***m***`)`	È un socket di dominio Unix?

Naturalmente, anche se nel file system possono esistere file di ogni tipo, poi os16 non è in grado di gestire i file FIFO, i collegamenti simbolici e nemmeno i socket di dominio Unix.

Nel file '`sys/stat.h`' sono definite anche delle macro-variabili per individuare e facilitare la selezione dei bit che compongono le informazioni del membro ***st_mode***:

Modalità simbolica	Modalità numerica	Descrizione
`S_IFMT`	0170000_8	Maschera che raccoglie tutti i bit che individuano il tipo di file.
`S_IFBLK`	0060000_8	File di dispositivo a blocchi.
`S_IFCHR`	0020000_8	File di dispositivo a caratteri.
`S_IFIFO`	0010000_8	File FIFO.
`S_IFREG`	0100000_8	File puro e semplice.
`S_IFDIR`	0040000_8	Directory.
`S_IFLNK`	0120000_8	Collegamento simbolico.
`S_IFSOCK`	0140000_8	Socket di dominio Unix.

Modalità simbolica	Modalità numerica	Descrizione
`S_ISUID`	0004000_8	SUID.
`S_ISGID`	0002000_8	SGID.
`S_ISVTX`	0001000_8	Sticky.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXU`	0000700_8	Lettura, scrittura ed esecuzione per l'utente proprietario.
`S_IRUSR`	0000400_8	Lettura per l'utente proprietario.
`S_IWUSR`	0000200_8	Scrittura per l'utente proprietario.
`S_IXUSR`	0000100_8	Esecuzione per l'utente proprietario.

Modalità simbolica	Modalità numerica	Descrizione
S_IRWXG	0000070_8	Lettura, scrittura ed esecuzione per il gruppo.
S_IRGRP	0000040_8	Lettura per il gruppo.
S_IWGRP	0000020_8	Scrittura per il gruppo.
S_IXGRP	0000010_8	Esecuzione per il gruppo.

Modalità simbolica	Modalità numerica	Descrizione
S_IRWXO	0000007_8	Lettura, scrittura ed esecuzione per gli altri utenti.
S_IROTH	0000004_8	Lettura per gli altri utenti.
S_IWOTH	0000002_8	Scrittura per gli altri utenti.
S_IXOTH	0000001_8	Esecuzione per gli altri utenti.

os16 non considera i permessi SUID (*Set user id*), SGID (*Set group id*) e Sticky; inoltre, non tiene in considerazione i permessi legati al gruppo, perché non ne tiene traccia.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
ENFILE	Troppi file aperti nel sistema.
ENOENT	File non trovato.
EACCES	Permesso negato.
EBADF	Il descrittore del file richiesto non è valido.

FILE SORGENTI

'`lib/sys/stat.h`' [101.13]

'`lib/sys/stat/stat.c`' [101.13.6]

'`lib/sys/stat/fstat.c`' [101.13.3]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/path_stat.c`' [100.4.44]

'`kernel/fs/fd_stat.c`' [100.4.11]

VEDERE ANCHE

chmod(2) [93.4], *chown(2)* [93.5].

93.37 os16: sys(2)

«

NOME

'**`sys`**' - chiamata di sistema

SINTASSI

```
#include <sys/os16.h>
void sys (int syscallnr, void *message, size_t size);
```

DESCRIZIONE

Attraverso la funzione ***sys()*** si effettuano tutte le chiamate di sistema, passando al kernel un messaggio, a cui punta ***message***, lungo ***size*** byte. A seconda dei casi, il messaggio può essere modificato dal kernel, come risposta alla chiamata.

Il messaggio è in pratica una variabile strutturata, la cui articolazione cambia a seconda del tipo di chiamata, pertanto si rende necessario specificarne ogni volta la dimensione.

Le strutture usate per comporre i messaggi hanno alcuni membri ricorrenti frequentemente:

Membro	Descrizione
`char path[PATH_MAX];`	Un percorso di file o directory.
`… ret;`	Serve a contenere il valore restituito dalla funzione che nel kernel compie effettivamente il lavoro. Il tipo del membro varia caso per caso.
`int errno;`	Serve a contenere il numero dell'errore prodotto dalla funzione che nel kernel compie effettivamente il lavoro.
`int errln;`	Serve a contenere il numero della riga di codice in cui si è prodotto un errore.
`char errfn[PATH_MAX];`	Serve a contenere il nome del file in cui si è prodotto un errore.

Le funzioni che si avvalgono di ***sys()***, prima della chiamata devono provvedere a compilare il messaggio con i dati necessari, dopo la chiamata devono acquisire i dati di ritorno, contenuti nel messaggio aggiornato dal kernel, e in particolare devono aggiornare le variabili ***errno***, ***errln*** e ***errfl***, utilizzando i membri con lo stesso nome presenti nel messaggio.

FILE SORGENTI

'`lib/sys/os16.h`' [101.12]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

93.38 os16: stime(2)

Vedere *time(2)* [93.39].

93.39 os16: time(2)

NOME

'**`time`**', '**`stime`**' - lettura o impostazione della data e dell'ora del sistema

SINTASSI

```
#include <time.h>
time_t time   (time_t *timer);
int    stime  (time_t *timer);
```

DESCRIZIONE

La funzioni ***time()*** legge la data e l'ora attuale del sistema, espressa in secondi; se il puntatore ***timer*** è valido (non è '**`NULL`**'), il risultato dell'interrogazione viene salvato anche in ciò a cui questo punta.

La funzione ***stime()*** consente di modificare la data e l'ora attuale del sistema, fornendo il puntatore alla variabile contenente la quantità di secondi trascorsi a partire dall'ora zero del 1 gennaio 1970.

VALORE RESTITUITO

La funzione ***time()*** restituisce la data e l'ora attuale del sistema, espressa in secondi trascorsi a partire dall'ora zero del 1 gennaio 1970.

La funzione ***stime()*** restituisce zero in caso di successo e, teoricamente, −1 in caso di errore, ma attualmente nessun tipo di errore è previsto.

DIFETTI

La funzione ***stime()*** dovrebbe essere riservata a un utente privilegiato, mentre attualmente qualunque utente può servirsene per cambiare la data di sistema.

FILE SORGENTI

'`lib/time.h`' [101.16]

'`lib/time/time.c`' [101.16.6]

'`lib/time/stime.c`' [101.16.5]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/k_libc.h`' [100.6]

'`kernel/k_libc/k_time.c`' [100.6.11]

'`kernel/k_libc/k_stime.c`' [100.6.10]

VEDERE ANCHE

date(1) [92.8], *ctime(3)* [94.13].

93.40 os16: umask(2)

NOME

'**`umask`**' - maschera dei permessi

SINTASSI

```
#include <sys/stat.h>
mode_t umask (mode_t mask);
```

DESCRIZIONE

La funzione ***umask()*** modifica la maschera dei permessi associata al processo in corso. Nel contenuto del parametro ***mask*** vengono presi in considerazione soltanto i nove bit meno significativi, i quali rappresentano i premessi di accesso di utente proprietario, gruppo e altri utenti.

La maschera dei permessi viene usata dalle funzioni che creano dei file, per limitare i permessi in modo automatico: ciò che appare attivo nella maschera è quello che non viene consentito nella creazione del file.

VALORE RESTITUITO

La funzione restituisce il valore che aveva la maschera dei permessi prima della chiamata.

FILE SORGENTI

'`lib/sys/stat.h`' [101.13]

'`lib/sys/stat/umask.c`' [101.13.7]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

VEDERE ANCHE

mkdir(2) [93.25], *chmod(2)* [93.4], *open(2)* [93.28], *stat(2)* [93.36].

93.41 os16: umount(2)

Vedere *mount(2)* [93.27].

93.42 os16: unlink(2)

NOME

'**`unlink`**' - cancellazione di un nome

SINTASSI

```
#include <unistd.h>
int unlink (const char *path);
```

DESCRIZIONE

La funzione ***unlink()*** cancella un nome da una directory, ma se si tratta dell'ultimo collegamento che ha quel file, allora libera anche l'inode corrispondente.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Errore: la variabile ***errno*** viene impostata di conseguenza.

ERRORI

Valore di ***errno***	Significato
`ENOTEMPTY`	È stata tentata la cancellazione di una directory, ma questa non è vuota.
`ENOTDIR`	Una delle directory del percorso, non è una directory.
`ENOENT`	Il nome richiesto non esiste.
`EROFS`	Il file system è in sola lettura.
`EPERM`	Mancano i permessi necessari.
`EUNKNOWN`	Si è verificato un errore imprevisto e sconosciuto.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/unlink.c`' [101.17.35]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/path_unlink.c`' [100.4.46]

VEDERE ANCHE

rm(1) [92.18], *link(2)* [93.23], *rmdir(2)* [93.30].

93.43 os16: wait(2)

NOME

'**wait**' - attesa della morte di un processo figlio

SINTASSI

```
#include <sys/wait.h>
pid_t wait (int *status);
```

DESCRIZIONE

La funzione ***wait()*** mette il processo in pausa, in attesa della morte di un processo figlio; quando ciò dovesse accadere, il valore di ****status*** verrebbe aggiornato con quanto restituito dal processo defunto e il processo sospeso riprenderebbe l'esecuzione.

VALORE RESTITUITO

La funzione restituisce il numero del processo defunto, oppure -1 se non ci sono processi figli.

ERRORI

Valore di ***errno***	Significato
`ECHILD`	Non ci sono processi figli da attendere.

FILE SORGENTI

'`lib/sys/wait.h`' [101.15]

'`lib/sys/wait/wait.c`' [101.15.1]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_wait.c`' [100.9.28]

VEDERE ANCHE

_exit(2) [93.2], *fork(2)* [93.14], *kill(2)* [93.22], *signal(2)* [93.34].

93.44 os16: write(2)

NOME

'**write**' - scrittura di un descrittore di file

SINTASSI

```
#include <unistd.h>
ssize_t write (int fdn, const void *buffer, size_t count);
```

DESCRIZIONE

La funzione ***write()*** consente di scrivere fino a un massimo di ***count*** byte, tratti dall'area di memoria che inizia all'indirizzo ***buffer***, presso il file rappresentato dal descrittore ***fdn***. La scrittura avviene a partire dalla posizione in cui si trova l'indice interno.

VALORE RESTITUITO

La funzione restituisce la quantità di byte scritti effettivamente e in tal caso è possibile anche ottenere una quantità pari a zero. Se si verifica invece un errore, la funzione restituisce −1 e aggiorna la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il numero del descrittore di file non è valido.
`EINVAL`	Il file non è aperto in scrittura.
`EISDIR`	Il file è una directory.
`E_FILE_TYPE_UNSUPPORTED`	Il file è di tipo non gestibile con os16.
`EIO`	Errore di input-output.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/write.c`' [101.17.36]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

'kernel/fs/fd_write.c' [100.4.12]

VEDERE ANCHE

close(2) [93.7], *lseek(2)* [93.24], *open(2)* [93.28], *read(2)* [93.29], *fwrite(3)* [94.48].

93.45 os16: z(2)

NOME

'**z_**...' - funzioni provvisorie

SINTASSI

```
#include <sys/os16.h>
void z_perror   (const char *string);
int  z_printf   (char *format, ...);
int  z_putchar  (int c);
int  z_puts     (char *string);
int  z_vprintf  (char *format, va_list arg);
```

DESCRIZIONE

Le funzioni del gruppo '**z_**...**()**' eseguono compiti equivalenti a quelli delle funzioni di libreria con lo stesso nome, ma prive del prefisso '**z_**'. Queste funzioni '**z_**...**()**' si avvalgono, per il loro lavoro, di chiamate di sistema particolari; la loro realizzazione si è resa necessaria durante lo sviluppo di os16, prima che potesse essere disponibile un sistema di gestione centralizzato dei dispositivi.

Queste funzioni non sono più utili, ma rimangono per documentare le fasi realizzative iniziali di os16 e, d'altro canto, possono servire se si rende necessario raggirare la gestione dei dispositivi per visualizzare dei messaggi sullo schermo.

FILE SORGENTI

'lib/sys/os16.h' [101.12]

'lib/sys/os16/z_perror.c' [101.12.17]

'lib/sys/os16/z_printf.c' [101.12.18]

'lib/sys/os16/z_putchar.c' [101.12.19]

'lib/sys/os16/z_puts.c' [101.12.20]

'lib/sys/os16/z_vprintf.c' [101.12.21]

VEDERE ANCHE

perror(3) [94.77], *printf(3)* [94.78], *putchar(3)* [94.37], *puts(3)* [94.38], *vprintf(3)* [94.128], *vsprintf(3)* [94.128].

93.46 os16: z_perror(2)

«

Vedere *z(2)* [93.45].

93.47 os16: z_printf(2)

«

Vedere *z(2)* [93.45].

93.48 os16: z_putchar(2)

«

Vedere *z(2)* [93.45].

93.49 os16: z_puts(2)

«

Vedere *z(2)* [93.45].

93.50 os16: z_vprintf(2)

«

Vedere *z(2)* [93.45].

Sezione 3: funzioni di libreria

Capitolo 94

TXT
htm
PDF
pdf
P1
P2
P3
P4
epub

94.1 os16: access(3)

NOME

'**access**' - verifica dei permessi di accesso dell'utente

SINTASSI

```
#include <unistd.h>
int access (const char *path, int mode);
```

DESCRIZIONE

La funzione ***access()*** verifica lo stato di accessibilità del file indicato nella stringa ***path***, secondo i permessi stabiliti con il parametro ***mode***.

L'argomento corrispondente al parametro ***mode*** può assumere un valore corrispondente alla macro-variabile ***F_OK***, per verificare semplicemente l'esistenza del file specificato; altrimenti, può avere un valore composto dalla combinazione (con l'operatore OR binario) di ***R_OK***, ***W_OK*** e ***X_OK***, per verificare, rispettivamente, l'accessibilità in lettura, in scrittura e in esecuzione. Queste macro-variabili sono dichiarate nel file '`unistd.h`'.

VALORE RESTITUITO

Valore	Significato
0	I permessi di accesso richiesti sono tutti disponibili.
−1	I permessi non sono tutti disponibili, oppure si è verificato un errore di altro genere, da chiarire analizzando la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`ENFILE`	Troppi file aperti nel sistema.
`ENOENT`	File non trovato.
`EACCES`	Permesso negato.

DIFETTI

Questa realizzazione della funzione ***access()*** determina l'accessibilità a un file attraverso le informazioni che può trarre autonomamente, senza usare una chiamata di sistema. Pertanto, si tratta di una valutazione presunta e non reale.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/access.c`' [101.17.2]

VEDERE ANCHE

stat(2) [93.36].

94.2 os16: abort(3)

«

NOME

'**`abort`**' - conclusione anormale del processo

SINTASSI

```
#include <stdlib.h>
void abort (void);
```

DESCRIZIONE

La funzione ***abort()*** verifica lo stato di configurazione del segnale '**`SIGABRT`**' e, se risulta bloccato, lo sblocca, quindi invia questo segnale per il processo in corso. Ciò provoca la conclusione del processo, secondo la modalità prevista per tale segnale, a meno che il segnale sia stato ridiretto a una funzione, nel qual caso, dopo l'invio del segnale, potrebbe esserci anche una ripresa del controllo da parte della funzione ***abort()***. Tuttavia, se così fosse, il segnale '**`SIGABRT`**' verrebbe poi riconfigurato alla sua impostazione normale e verrebbe inviato nuovamente lo stesso segnale per provocare la conclusione del processo. Pertanto, la funzione ***abort()*** non restituisce il controllo.

Va comunque osservato che os16 non è in grado di associare una funzione a un segnale, pertanto, i segnali possono solo avere una gestione predefinita, o al massimo risultare bloccati.

FILE SORGENTI

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/abort.c`' [101.10.2]

VEDERE ANCHE

signal(2) [93.34].

94.3 os16: abs(3)

NOME

'**abs**', '**labs**' - valore assoluto di un numero intero

SINTASSI

```
#include <stdlib.h>
int abs (int j);
long int labs (long int j);
```

DESCRIZIONE

Le funzioni '...**abs()**' restituiscono il valore assoluto del loro argomento. Si distinguono per tipo di intero e, nel caso di os16, non essendo disponibile il tipo '**long long int**', si limitano a ***abs()*** e ***labs()***.

VALORE RESTITUITO

Il valore assoluto del numero intero fornito come argomento.

FILE SORGENTI

'lib/stdlib.h' [101.10]

'lib/stdlib/abs.c' [101.10.3]

'lib/stdlib/labs.c' [101.10.12]

VEDERE ANCHE

div(3) [94.15], *ldiv(3)* [94.15], *rand(3)* [94.85].

94.4 os16: atexit(3)

NOME

'**atexit**', '**exit**' - gestione della chiusura dei processi

SINTASSI

```
#include <stdlib.h>
typedef void (*atexit_t) (void);
int  atexit (atexit_t function);
void exit   (int status);
```

DESCRIZIONE

La funzione ***exit()*** conclude il processo in corso, avvalendosi della chiamata di sistema *_exit(2)* [93.2], ma prima di farlo, scandisce un array contenente un elenco di funzioni, da eseguire prima di tale chiamata finale. Questo array viene popolato eventualmente con l'aiuto della funzione ***atexit()***, sapendo che l'ultima funzione di chiusura aggiunta, è la prima a dover essere eseguita alla conclusione.

La funzione ***atexit()*** riceve come argomento il puntatore a una funzione che non prevede argomenti e non restituisce alcunché. Per facilitare la dichiarazione del prototipo e, di conseguenza, dell'array usato per accumulare tali puntatori, il file di intestazione '`stdlib.h`' di os16 dichiara un tipo speciale, non standard, denominato '**`atexit_t`**', definito come:

```
typedef void (*atexit_t) (void);
```

Si possono annotare un massimo di ***ATEXIT_MAX*** funzioni da eseguire prima della conclusione di un processo. Tale macro-variabile è definita nel file '`limits.h`'.

VALORE RESTITUITO

Solo la funzione ***atexit()*** restituisce un valore, perché ***exit()*** non può nemmeno restituire il controllo.

Valore	Significato
0	Operazione conclusa con successo.
−1	Errore, dovuto all'esaurimento dello spazio nell'array usato per accumulare le funzioni di chiusura.

FILE SORGENTI

'`lib/limits.h`' [101.1.8]

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/atexit.c`' [101.10.5]

'`lib/stdlib/exit.c`' [101.10.10]

VEDERE ANCHE

_exit(2) [93.2], *_Exit(2)* [93.2].

94.5 os16: atoi(3)

NOME

'**`atoi`**', '**`atol`**' - conversione da stringa a numero intero

SINTASSI

```
#include <stdlib.h>
int atoi (const char *string);
long int atol (const char *string);
```

DESCRIZIONE

Le funzioni '**ato...()**' convertono una stringa, fornita come argomento, in un numero intero. La conversione avviene escludendo gli spazi iniziali, considerando eventualmente un segno («+» o «−») e poi soltanto i caratteri che rappresentano cifre numeriche. La scansione della stringa e l'interpretazione del valore numerico contenuto terminano quando si incontra un carattere diverso dalle cifre numeriche.

VALORE RESTITUITO

Il valore numerico ottenuto dall'interpretazione della stringa.

FILE SORGENTI

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/atoi.c`' [101.10.6]

'`lib/stdlib/atol.c`' [101.10.7]

VEDERE ANCHE

strtol(3) [94.121], *strtoul(3)* [94.121].

94.6 os16: atol(3)

Vedere *atoi(3)* [94.5].

94.7 os16: basename(3)

NOME

'**basename**', '**dirname**' - elaborazione dei componenti di un percorso

SINTASSI

```
#include <libgen.h>
char *basename (char *path);
char *dirname (char *path);
```

DESCRIZIONE

Le funzioni ***basename()*** e ***dirname()***, restituiscono un percorso, estratto da quello fornito come argomento (***path***). Per la precisione, ***basename()*** restituisce l'ultimo componente del percorso, mentre ***dirname()*** restituisce ciò che precede l'ultimo componente. Valgono gli esempi seguenti:

Contenuto originale di ***path***	Risultato prodotto da '**`dirname(`*path*`)`**'	Risultato prodotto da '**`basename(`*path*`)`**'
`"/usr/bin/`	`"/usr"`	`"bin"`
`"/usr/bin`	`"/usr"`	`"bin"`
`"/usr`	`"/"`	`"usr"`
`"usr`	`"."`	`"usr"`
`"/"`	`"/"`	`"/"`
`"."`	`"."`	`"."`
`".."`	`".."`	`".."`

È importante considerare che le due funzioni alterano il contenuto di ***path***, in modo da isolare i componenti che servono.

VALORE RESTITUITO

Le due funzioni restituiscono il puntatore alla stringa contenente il risultato dell'elaborazione, trattandosi di una porzione della stringa già usata come argomento della chiamata e modificata per l'occasione. Non è previsto il manifestarsi di alcun errore.

FILE SORGENTI

'`lib/libgen.h`' [101.6]

'`lib/libgen/basename.c`' [101.6.1]

'`lib/libgen/dirname.c`' [101.6.2]

94.8 os16: bp(3)

Vedere *cs(3)* [94.12].

94.9 os16: clearerr(3)

NOME

'**clearerr**' - azzeramento degli indicatori di errore e di fine file di un certo flusso di file

SINTASSI

```
#include <stdio.h>
void clearerr (FILE *fp);
```

DESCRIZIONE

La funzione ***clearerr()*** azzera gli indicatori di errore e di fine file, del flusso di file indicato come argomento.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/clearerr.c`' [101.9.2]

VEDERE ANCHE

feof(3) [94.28], *ferror(3)* [94.29], *fileno(3)* [94.34], *stdio(3)* [94.103].

94.10 os16: closedir(3)

NOME

'**closedir**' - chiusura di una directory

SINTASSI

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dp);
```

DESCRIZIONE

La funzione ***closedir()*** chiude la directory rappresentata da ***dp***.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Errore: la variabile ***errno*** viene impostata di conseguenza.

ERRORI

Valore di ***errno***	Significato
`EBADF`	La directory rappresentata da ***dp***, non è valida.

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/dirent.h`' [101.2]

'`lib/dirent/DIR.c`' [101.2.1]

'`lib/dirent/closedir.c`' [101.2.2]

VEDERE ANCHE

close(2) [93.7], *opendir(3)* [94.76], *readdir(3)* [94.86], *rewinddir(3)* [94.89].

94.11 os16: creat(3)

NOME

'**`creat`**' - creazione di un file puro e semplice

SINTASSI

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *path, mode_t mode);
```

DESCRIZIONE

La funzione ***creat()*** equivale esattamente all'uso della funzione ***open()***, con le opzioni '**`O_WRONLY|O_CREAT|O_TRUNC`**':

```
open (path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

Per ogni altra informazione, si veda la pagina di manuale *open(2)* [93.28].

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/sys/stat.h`' [101.13]

'`lib/fcntl.h`' [101.4]

'`lib/fcntl/creat.c`' [101.4.1]

VEDERE ANCHE

chmod(2) [93.4], *chown(2)* [93.5], *close(2)* [93.7], *dup(2)* [93.8], *fcntl(2)* [93.13], *link(2)* [93.24], *mknod(2)* [93.26], *mount(2)* [93.27], *open(2)* [93.28] *read(2)* [93.29], *stat(2)* [93.36], *umask(2)* [93.40], *unlink(2)* [93.42], *write(2)* [93.44], *fopen(3)* [94.35].

94.12 os16: cs(3)

«

NOME

'**`bp`**', '**`cs`**', '**`ds`**', '**`es`**', '**`sp`**', '**`ss`**' - stato dei registri della CPU

SINTASSI

```
#include <sys/os16.h>
unsigned int cs (void);
unsigned int ds (void);
unsigned int ss (void);
unsigned int es (void);
unsigned int sp (void);
unsigned int bp (void);
```

DESCRIZIONE

Le funzioni elencate nel quadro sintattico, sono in realtà delle macro-istruzioni, chiamanti funzioni con nomi analoghi, ma preceduti da un trattino basso (***_bp()***, ***_cs()***, ***_ds()***, ***_es()***, ***_sp()***, ***_ss()***), per interrogare lo stato di alcuni registri della CPU a fini diagnostici. I registri interessati sono quelli con lo stesso nome della macro-istruzione usata per interrogarli.

FILE SORGENTI

'`lib/sys/os16.h`' [101.12]

'`lib/sys/os16/_cs.s`' [101.12.2]

'`lib/sys/os16/_ds.s`' [101.12.3]

‘lib/sys/os16/_ss.s’ [101.12.8]

‘lib/sys/os16/_es.s’ [101.12.4]

‘lib/sys/os16/_sp.s’ [101.12.7]

‘lib/sys/os16/_bp.s’ [101.12.1]

VEDERE ANCHE

seg_i(3) [94.91].

seg_d(3) [94.91].

94.13 os16: ctime(3)

«

NOME

‘**asctime**’, ‘**ctime**’, ‘**gmtime**’, ‘**localtime**’, ‘**mktime**’ - conversione di informazioni data-orario

SINTASSI

```
#include <time.h>
char       *asctime   (const struct tm *timeptr);
char       *ctime     (const time_t *timer);
struct tm *gmtime     (const time_t *timer);
struct tm *localtime (const time_t *timer);
time_t      mktime    (const struct tm *timeptr);
```

DESCRIZIONE

Queste funzioni hanno in comune il compito di convertire delle informazioni data-orario, da un formato a un altro, eventualmente anche testuale. Una data e un orario possono essere rappresentati con il tipo ‘**time_t**’, nel qual caso si tratta del numero di secondi trascorsi dall’ora zero del 1 gennaio 1970; in alternativa potrebbe essere rappresentata in una variabile strutturata di tipo ‘**struct tm**’, dichiarato nel file ‘time.h’:

```
struct tm {
    int tm_sec;      // secondi
    int tm_min;      // minuti
    int tm_hour;     // ore
    int tm_mday;     // giorno del mese
    int tm_mon;      // mese, da 1 a 12
    int tm_year;     // anno
    int tm_wday;     // giorno della settimana,
                     // da 0 (domenica) a 6
    int tm_yday;     // giorno dell'anno
    int tm_isdst;    // informazioni sull'ora estiva
};
```

In alcuni casi, la conversione dovrebbe tenere conto della configurazione locale, ovvero del fuso orario ed eventualmente del cambiamento di orario nel periodo estivo. os16 non considera alcunché e gestisce il tempo in un modo assoluto, senza nozione della convenzione locale.

La funzione ***asctime()*** converte quanto contenuto in una variabile strutturata di tipo '**`struct tm`**' in una stringa che descrive la data e l'ora in inglese. La stringa in questione è allocata staticamente e viene sovrascritta se la funzione viene usata più volte.

La funzione ***ctime()*** è in realtà soltanto una macro-istruzione, la quale, complessivamente, converte il tempo indicato come quantità di secondi, restituendo il puntatore a una stringa che descrive la data attuale, tenendo conto della configurazione locale. La stringa in questione utilizza la stessa memoria statica usata per ***asctime()***; inoltre, dato che os16 non distingue tra ora locale e tempo universale, la funzione non esegue alcuna conversione temporale.

La funzione ***gmtime()*** converte il tempo espresso in secondi in una forma suddivisa, secondo il tipo '**`struct tm`**'. La funzione restituisce quindi il puntatore a una variabile strutturata di tipo '**`struct tm`**', la quale però è dichiarata in modo statico, internamente alla funzione, e viene sovrascritta nelle chiamate successive della stessa.

La funzione ***localtime()*** converte una data espressa in secondi, in una data suddivisa in campi ('**`struct tm`**'), tenendo conto (ma non succede con os16) della configurazione locale.

La funzione ***mktime()*** converte una data contenute in una variabile strutturata di tipo '**`struct tm`**' nella quantità di secondi corrispondente.

VALORE RESTITUITO

Le funzioni che restituiscono un puntatore, se incontrano un errore, restituiscono il puntatore nullo, '**`NULL`**'. Nel caso particolare di ***mktime()***, se il valore restituito è pari a −1, si tratta di un errore.

FILE SORGENTI

'`lib/time.h`' [101.16]

'`lib/time/asctime.c`' [101.16.1]

'`lib/time/gmtime.c`' [101.16.3]

'`lib/time/mktime.c`' [101.16.4]

VEDERE ANCHE

date(1) [92.8], *clock(2)* [93.6], *time(2)* [93.39].

94.14 os16: dirname(3)

Vedere *basename(3)* [94.7].

94.15 os16: div(3)

NOME

'**`div`**', '**`ldiv`**' - calcolo del quoziente e del resto di una divisione intera

SINTASSI

```
#include <stdlib.h>
div_t div (int numer, int denom);
ldiv_t ldiv (long int numer, long int denom);
```

DESCRIZIONE

Le funzioni '...**`div()`**' calcolano la divisione tra numeratore e denominatore, forniti come argomenti della chiamata, restituendo un risultato, composto di divisione intera e resto, in una variabile strutturata.

I tipi '**`div_t`**' e '**`ldiv_t`**', sono dichiarati nel file '`stdlib.h`' nel modo seguente:

```
typedef struct {
    int      quot;
    int      rem;
} div_t;
//
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
```

I membri ***quot*** contengono il quoziente, ovvero il risultato intero; i membri ***rem*** contengono il resto della divisione.

VALORE RESTITUITO

Il risultato della divisione, strutturato in quoziente e resto.

FILE SORGENTI

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/div.c`' [101.10.8]

'`lib/stdlib/ldiv.c`' [101.10.13]

VEDERE ANCHE

abs(3) [94.3].

94.16 os16: ds(3)

Vedere *cs(3)* [94.12].

94.17 os16: endpwent(3)

Vedere *getpwent(3)* [94.53].

94.18 os16: errno(3)

NOME

'**`errno`**' - numero dell'ultimo errore riportato

SINTASSI

```
#include <errno.h>
```

DESCRIZIONE

Attraverso l'inclusione del file '`errno.h`', si ottiene la dichiarazione della variabile ***errno***. In pratica, per os16 viene dichiarata così:

```
extern int errno;
```

Per annotare un errore, si assegna un valore numerico a questa variabile. Il valore numerico in questione rappresenta sinteticamente la descrizione dell'errore; pertanto, si utilizzano per questo delle macro-variabili, dichiarate tutte nel file '`errno.h`'. Nell'esempio seguente si annota l'errore ***ENOMEM***, corrispondente alla descrizione «Not enough space», ovvero «spazio insufficiente»:

```
errno = ENOMEM;
```

Dal momento che in questo modo viene comunque a mancare un riferimento al sorgente e alla posizione in cui l'errore si è manifestato, la libreria di os16 aggiunge la macro-istruzione ***errset()***, la quale però non fa parte dello standard. Si usa così:

```
errset (ENOMEM);
```

La macro-istruzione ***errset()*** aggiorna la variabile ***errln*** e l'array ***errfn[]***, rispettivamente con il numero della riga di codice in cui si è manifestato il problema e il nome della funzione che lo contiene. Con queste informazioni, la funzione *perror(3)* [94.77] può visualizzare più dati.

Pertanto, nel codice di os16, si usa sempre la macro-istruzione ***errset()***, invece di assegnare semplicemente un valore alla variabile ***errno***.

ERRORI

Gli errori previsti dalla libreria di os16 sono riassunti dalla tabella successiva. La prima parte contiene gli errori definiti dallo standard POSIX, ma solo alcuni di questi vengono usati effettivamente nella libreria, data la limitatezza di os16.

Valore di ***errno***	Definizione
`E2BIG`	Argument list too long.
`EACCES`	Permission denied.
`EADDRINUSE`	Address in use.
`EADDRNOTAVAIL`	Address not available.
`EAFNOSUPPORT`	Address family not supported.

Valore di ***errno***	Definizione
`EAGAIN`	Resource unavailable, try again.
`EALREADY`	Connection already in progress.
`EBADF`	Bad file descriptor.
`EBADMSG`	Bad message.
`EBUSY`	Device or resource busy.

Valore di ***errno***	Definizione
`ECANCELED`	Operation canceled.
`ECHILD`	No child processes.
`ECONNABORTED`	Connection aborted.
`ECONNREFUSED`	Connection refused.
`ECONNRESET`	Connection reset.

Valore di *errno*	Definizione
`EDEADLK`	Resource deadlock would occur.
`EDESTADDRREQ`	Destination address required.
`EDOM`	Mathematics argument out of domain of function.
`EDQUOT`	Reserved.
`EEXIST`	File exists.

Valore di *errno*	Definizione
`EFAULT`	Bad address.
`EFBIG`	File too large.
`EHOSTUNREACH`	Host is unreachable.
`EIDRM`	Identifier removed.
`EILSEQ`	Illegal byte sequence.

Valore di *errno*	Definizione
`EINPROGRESS`	Operation in progress.
`EINTR`	Interrupted function.
`EINVAL`	Invalid argument.
`EIO`	I/O error.
`EISCONN`	Socket is connected.

Valore di *errno*	Definizione
`EISDIR`	Is a directory.
`ELOOP`	Too many levels of symbolic links.
`EMFILE`	Too many open files.
`EMLINK`	Too many links.
`EMSGSIZE`	Message too large.

Valore di *errno*	Definizione
`EMULTIHOP`	Reserved.
`ENAMETOOLONG`	Filename too long.
`ENETDOWN`	Network is down.
`ENETRESET`	Connection aborted by network.
`ENETUNREACH`	Network unreachable.

Valore di *errno*	Definizione
`ENFILE`	Too many files open in system.
`ENOBUFS`	No buffer space available.
`ENODATA`	No message is available on the stream head read queue.
`ENODEV`	No such device.
`ENOENT`	No such file or directory.

Valore di *errno*	Definizione
`ENOEXEC`	Executable file format error.
`ENOLCK`	No locks available.
`ENOLINK`	Reserved.
`ENOMEM`	Not enough space.
`ENOMSG`	No message of the desired type.

Valore di *errno*	Definizione
`ENOPROTOOPT`	Protocol not available.
`ENOSPC`	No space left on device.
`ENOSR`	No stream resources.
`ENOSTR`	Not a stream.
`ENOSYS`	Function not supported.

Valore di *errno*	Definizione
`ENOTCONN`	The socket is not connected.
`ENOTDIR`	Not a directory.
`ENOTEMPTY`	Directory not empty.
`ENOTSOCK`	Not a socket.
`ENOTSUP`	Not supported.

Valore di ***errno***	Definizione
`ENOTTY`	Inappropriate I/O control operation.
`ENXIO`	No such device or address.
`EOPNOTSUPP`	Operation not supported on socket.
`EOVERFLOW`	Value too large to be stored in data type.
`EPERM`	Operation not permitted.
`EPIPE`	Broken pipe.
`EPROTO`	Protocol error.
`EPROTONOSUPPORT`	Protocol not supported.
`EPROTOTYPE`	Protocol wrong type for socket.
`ERANGE`	Result too large.
`EROFS`	Read-only file system.
`ESPIPE`	Invalid seek.
`ESRCH`	No such process.
`ESTALE`	Reserved.
`ETIME`	Stream ioctl() timeout.

Valore di *errno*	Definizione
ETIMEDOUT	Connection timed out.
ETXTBSY	Text file busy.
EWOULDBLOCK	Operation would block (may be the same as EAGAIN).
EXDEV	Cross-device link.

La tabella successiva raccoglie le definizioni degli errori aggiuntivi, specifici di os16.

Valore di *errno*	Definizione
EUNKNOWN	Unknown error.
E_FILE_TYPE	File type not compatible.
E_ROOT_INODE_NOT_CACHED	The root directory inode is not cached.
E_CANNOT_READ_SUPERBLOCK	Cannot read super block.
E_MAP_INODE_TOO_BIG	Map inode too big.
E_MAP_ZONE_TOO_BIG	Map zone too big.
E_DATA_ZONE_TOO_BIG	Data zone too big.
E_CANNOT_FIND_ROOT_DEVICE	Cannot find root device.
E_CANNOT_FIND_ROOT_INODE	Cannot find root inode.
E_FILE_TYPE_UNSUPPORTED	File type unsupported.

Valore di ***errno***	Definizione
`E_ENV_TOO_BIG`	Environment too big.
`E_LIMIT`	Exceeded implementation limits.
`E_NOT_MOUNTED`	Not mounted.
`E_NOT_IMPLEMENTED`	Not implemented.

FILE SORGENTI

'`lib/errno.h`' [101.3]

'`lib/errno/errno.c`' [101.3.1]

VEDERE ANCHE

perror(3) [94.77], *strerror(3)* [94.111].

94.19 os16: es(3)

Vedere *cs(3)* [94.12].

94.20 os16: exec(3)

NOME

'**`execl`**', '**`execle`**', '**`execlp`**', '**`execv`**', '**`execvp`**' - esecuzione di un file

SINTASSI

```
#include <unistd.h>
extern char **environ;
int execl  (const char *path, const char *arg, ...);
int execle (const char *path, const char *arg, ...);
int execlp (const char *path, const char *arg, ...);
int execv  (const char *path, char *const argv[]);
int execvp (const char *path, char *const argv[]);
```

DESCRIZIONE

Le funzioni '`exec...()`' rimpiazzano il processo chiamante con un altro processo, ottenuto dal caricamento di un file eseguibile in memoria. Tutte queste funzioni si avvalgono, direttamente o indirettamente di *execve(2)* [93.10].

Il primo argomento delle funzioni descritte qui è il percorso, rappresentato dalla stringa ***path***, di un file da eseguire.

Le funzioni '`execl...()`', dopo il percorso del file eseguibile, richiedono l'indicazione di una quantità variabile di argomenti, a cominciare da ***arg***, costituiti da stringhe, tenendo conto che dopo l'ultimo di questi argomenti, occorre fornire il puntatore nullo, '`NULL`', per chiarire che questi sono terminati. L'argomento corrispondente al parametro ***arg*** deve essere una stringa contenente il nome del file da avviare, mentre gli argomenti successivi sono gli argomenti da passare al programma stesso.

Rispetto al gruppo di funzioni '`execl...()`', la funzione ***execle()***, dopo il puntatore nullo che conclude la sequenza di argomenti per il programma da avviare, si attende una sequenza di altre stringhe, anche questa conclusa da un ultimo e definitivo puntatore nullo. Questa ulteriore sequenza di stringhe va a costituire l'ambiente del processo da avviare, pertanto il contenuto di tali stringhe deve essere del tipo '***nome=valore***'.

Le funzioni '`execv...()`' hanno come ultimo argomento il puntatore a un array di stringhe, dove ***argv[0]*** deve essere il nome del file da avviare, mentre da ***argv[1]*** in poi, si tratta degli argomenti da passare al programma. Anche in questo caso, per riconoscere l'ultimo elemento di questo array, gli si deve assegnare il puntatore nullo.

Le funzioni ***execlp()*** e ***execvp()***, se ricevono solo il nome del file da eseguire, senza altre indicazioni del percorso, cercano questo file tra i percorsi indicati nella variabile di ambiente ***PATH***, ammesso che sia dichiarata per il processo in corso.

Tutte le funzioni qui descritte, a eccezione di ***execle()***, trasmettono al nuovo processo lo stesso ambiente (le stesse variabili di ambiente) del processo chiamante.

VALORE RESTITUITO

Queste funzioni, se hanno successo nel loro compito, non possono restituire alcunché, dato che in quel momento, il processo chiamante viene rimpiazzato da quello del file che viene eseguito. Pertanto, queste funzioni possono restituire soltanto un valore che rappresenta un errore, ovvero -1, aggiornando anche la variabile ***errno*** di conseguenza.

ERRORI

Valore di ***errno***	Significato
`E2BIG`	Ci sono troppi argomenti.
`ENOMEM`	Memoria insufficiente.
`ENOENT`	Il file richiesto non esiste.
`EACCES`	Il file non può essere avviato per la mancanza dei permessi di acceso necessari.
`ENOEXEC`	Il file non può essere un file eseguibile, perché non ne ha le caratteristiche.
`EIO`	Errore di input-output.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/execl.c`' [101.17.9]

'`lib/unistd/execle.c`' [101.17.10]

'`lib/unistd/execlp.c`' [101.17.11]

'`lib/unistd/execv.c`' [101.17.12]

'`lib/unistd/execvp.c`' [101.17.14]

'`lib/unistd/execve.c`' [101.17.13]

VEDERE ANCHE

execve(2) [93.10], *fork(2)* [93.14], *environ(7)* [97.1].

94.21 os16: execl(3)

Vedere *exec(3)* [94.20].

94.22 os16: execle(3)

Vedere *exec(3)* [94.20].

94.23 os16: execlp(3)

Vedere *exec(3)* [94.20].

94.24 os16: execv(3)

Vedere *exec(3)* [94.20].

94.25 os16: execvp(3)

Vedere *exec(3)* [94.20].

94.26 os16: exit(3)

Vedere *atexit(3)* [94.4].

94.27 os16: fclose(3)

NOME

'**`fclose`**' - chiusura di un flusso di file

SINTASSI

```
#include <stdio.h>
int fclose (FILE *fp);
```

DESCRIZIONE

La funzione ***fclose()*** chiude il flusso di file specificato tramite il puntatore ***fp***. Questa realizzazione particolare di os16, si limita a richiamare la funzione ***close()***, con l'indicazione del descrittore di file corrispondente al flusso.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
'**`EOF`**'	Errore: la variabile ***errno*** viene impostata di conseguenza.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file associato al flusso da chiudere, non è valido.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/fclose.c`' [101.9.3]

VEDERE ANCHE

close(2) [93.7], *fopen(3)* [94.35], *stdio(3)* [94.103].

94.28 os16: feof(3)

NOME

'**`feof`**' - verifica dello stato dell'indicatore di fine file

SINTASSI

```
#include <stdio.h>
int feof (FILE *fp);
```

DESCRIZIONE

La funzione ***feof()*** restituisce il valore dell'indicatore di fine file, riferito al flusso di file rappresentato da ***fp***.

VALORE RESTITUITO

Valore	Significato
diverso da zero	Significa che l'indicatore di fine file è impostato.
0	Significa che l'indicatore di fine file non è impostato.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/feof.c`' [101.9.4]

VEDERE ANCHE

clearerr(3) [94.9], *ferror(3)* [94.29], *fileno(3)* [94.34], *stdio(3)* [94.103].

94.29 os16: ferror(3)

NOME

'**`ferror`**' - verifica dello stato dell'indicatore di errore

SINTASSI

```
#include <stdio.h>
int ferror (FILE *fp);
```

DESCRIZIONE

La funzione ***ferror()*** restituisce il valore dell'indicatore di errore, riferito al flusso di file rappresentato da ***fp***.

VALORE RESTITUITO

Valore	Significato
diverso da zero	Significa che l'indicatore di errore è impostato.
0	Significa che l'indicatore di errore non è impostato.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/ferror.c`' [101.9.5]

VEDERE ANCHE

clearerr(3) [94.9], *feof(3)* [94.28], *fileno(3)* [94.34], *stdio(3)* [94.103].

94.30 os16: fflush(3)

«

NOME

'**`fflush`**' - fissaggio dei dati ancora sospesi nella memoria tampone

SINTASSI

```
#include <stdio.h>
int fflush (FILE *fp);
```

DESCRIZIONE

La funzione ***fflush()*** di os16, non fa alcunché, dato che non è prevista alcuna gestione della memoria tampone per i flussi di file.

VALORE RESTITUITO

Valore	Significato
0	Rappresenta il successo dell'operazione.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/fflush.c`' [101.9.6]

VEDERE ANCHE

fclose(3) [94.27], *fopen(3)* [94.35].

94.31 os16: fgetc(3)

NOME

'**`fgetc`**', '**`getc`**', '**`getchar`**' - lettura di un carattere da un flusso di file

SINTASSI

```
#include <stdio.h>
int fgetc   (FILE *fp);
int getc    (FILE *fp);
int getchar (void);
```

DESCRIZIONE

Le funzioni ***fgetc()*** e ***getc()*** sono equivalenti e leggono il carattere successivo dal flusso di file rappresentato da ***fp***. La funzione ***getchar()*** esegue la lettura di un carattere, ma dallo standard input.

VALORE RESTITUITO

In caso di successo, il carattere letto viene restituito in forma di intero positivo (il carattere viene inteso inizialmente senza segno, quindi viene trasformato in un intero, il quale rappresenta così un valore positivo). Se la lettura non può avere luogo, la funzione restituisce '**`EOF`**', corrispondente a un valore negativo.

ERRORI

La variabile ***errno*** potrebbe risultare aggiornata nel caso la funzione restituisca '**`EOF`**'. Ma per saperlo, occorre azzerare la variabile ***errno*** prima della chiamata di ***fgetc()***.

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file associato al flusso da cui leggere un carattere, non è valido.
`EINVAL`	Il descrittore di file associato al flusso da cui leggere un carattere, non consente un accesso in lettura.

FILE SORGENTI

'lib/stdio.h' [101.9]

'lib/stdio/fgetc.c' [101.9.7]

'lib/stdio/getchar.c' [101.9.24]

VEDERE ANCHE

fgets(3) [94.33], *gets(3)* [94.33].

94.32 os16: fgetpos(3)

«

NOME

'**fgetpos**', '**fsetpos**' - lettura e impostazione della posizione corrente di un flusso di file

SINTASSI

```
#include <stdio.h>
int fgetpos (FILE *restrict fp, fpos_t *restrict pos);
int fsetpos (FILE *restrict fp, fpos_t *restrict pos);
```

DESCRIZIONE

Le funzioni ***fgetpos()*** e ***fsetpos()***, rispettivamente, leggono o impostano la posizione corrente di un flusso di file.

Per os16, il tipo '**fpos_t**' è dichiarato nel file 'stdio.h' come equivalente al tipo '**off_t**' (file 'sys/types.h'); tuttavia, la modifica di una variabile di tipo '**fpos_t**' va fatta utilizzando una funzione apposita, perché lo standard consente che possa trattarsi anche di una variabile strutturata, i cui membri non sono noti.

L'uso della funzione ***fgetpos()*** comporta una modifica dell'informazione contenuta all'interno di ****pos***, mentre la funzione ***fsetpos()*** usa il valore contenuto in ****pos*** per cambiare la posizione corrente del flusso di file. In pratica, si può usare ***fsetpos()*** solo dopo aver salvato una certa posizione con l'aiuto di ***fgetpos()***.

Si comprende che il valore restituito dalle funzioni è solo un indice del successo o meno dell'operazione, dato che l'informazione sulla posizione viene ottenuta dalla modifica di una variabile di cui si fornisce il puntatore negli argomenti.

VALORE RESTITUITO

Valore	Significato
0	Rappresenta il successo dell'operazione.
−1	Indica il verificarsi di un errore, il quale può essere interpretato leggendo la variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EBADF`	Il descrittore di file associato al flusso, non è valido.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/fgetpos.c`' [101.9.8]

'`lib/stdio/fsetpos.c`' [101.9.20]

VEDERE ANCHE

fseek(3) [94.43], *ftell(3)* [94.46], *rewind(3)* [94.88].

94.33 os16: fgets(3)

NOME

'**`fgets`**', '**`gets`**' - lettura di una stringa da un flusso di file

SINTASSI

```
#include <stdio.h>
char *fgets (char *restrict string, int n,
             FILE *restrict fp);
char *gets  (char *string);
```

DESCRIZIONE

La funzione ***fgets()*** legge una «riga» dal flusso di file ***fp***, purché non più lunga di ***n***−1 caratteri, collocandola a partire da ciò a cui punta ***stringa***. La lettura termina al raggiungimento del carattere '**`\n`**' (*new line*), oppure alla fine del file, oppure a ***n***−1 caratteri. In ogni caso, viene aggiunto al termine, il codice nullo di terminazione di stringa: '**`\0`**'.

La funzione ***gets()***, in modo analogo a ***fgets()***, legge una riga dallo standard input, ma senza poter porre un limite massimo alla lunghezza della lettura.

VALORE RESTITUITO

In caso di successo, viene restituito il puntatore alla stringa contenente la riga letta, ovvero restituiscono ***string***. In caso di errore, o comunque in caso di una lettura nulla, si ottiene '**`NULL`**'. La variabile ***errno*** viene aggiornata solo se si

presenta un errore di accesso al file, mentre una lettura nulla, perché il flusso si è concluso, non comporta tale aggiornamento.

ERRORI

La variabile ***errno*** potrebbe risultare aggiornata nel caso le funzioni restituiscano '**NULL**'. Ma per saperlo, occorre azzerare la variabile ***errno*** prima della chiamata di queste.

Valore di ***errno***	Significato
EBADF	Il descrittore di file associato al flusso da cui leggere un carattere, non è valido.
EINVAL	Il descrittore di file associato al flusso da cui leggere un carattere, non consente un accesso in lettura.

FILE SORGENTI

'lib/stdio.h' [101.9]

'lib/stdio/fgets.c' [101.9.9]

'lib/stdio/gets.c' [101.9.25]

VEDERE ANCHE

fgetc(3) [94.31], *getc(3)* [94.31].

94.34 os16: fileno(3)

NOME

'**fileno**' - traduzione di un flusso di file nel numero di descrittore corrispondente

SINTASSI

```
#include <stdio.h>
int fileno (FILE *fp);
```

DESCRIZIONE

La funzione ***fileno()*** traduce il flusso di file, rappresentato da ***fp***, nel numero del descrittore corrispondente. Tuttavia, il risultato è valido solo se il flusso di file specificato è aperto effettivamente.

VALORE RESTITUITO

Se ***fp*** punta effettivamente a un flusso di file aperto, il valore restituito corrisponde al numero di descrittore del file stesso; diversamente, si potrebbe ottenere un

numero privo di senso. Se come argomento si indica il puntatore nullo, si ottiene un errore, rappresentato dal valore -1.

ERRORI

Valore di *errno*	Significato
`EBADF`	È stato richiesto di risolvere il puntatore nullo.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/FILE.c`' [101.9.1]

'`lib/stdio/fileno.c`' [101.9.10]

VEDERE ANCHE

clearerr(3) [94.9], *feof(3)* [94.28], *ferror(3)* [94.29], *stdio(3)* [94.103].

94.35 os16: fopen(3)

«

NOME

'**`fopen`**', '**`freopen`**' - apertura di un flusso di file

SINTASSI

```
#include <stdio.h>
FILE *fopen   (const char *path, const char *mode);
FILE *freopen (const char *restrict path,
               const char *restrict mode,
               FILE *restrict fp);
```

DESCRIZIONE

La funzione ***fopen()*** apre il file indicato nella stringa a cui punta ***path***, secondo le modalità di accesso contenute in ***mode***, associandovi un flusso di file. In modo analogo agisce anche la funzione ***freopen()***, la quale però, prima, chiude il flusso ***fp***.

La modalità di accesso al file si specifica attraverso una stringa, come sintetizzato dalla tabella successiva.

mode	Significato
`"r"` `"rb"`	Si richiede un accesso in lettura, di un file già esistente. L'indice interno per l'accesso ai dati viene posizionato all'inizio del file.

`"r+"`
`"r+b"`
`"rb+"`
	Si richiede un accesso in lettura e scrittura, di un file già esistente. L'indice interno per l'accesso ai dati viene posizionato all'inizio del file.

`"w"` `"wb"`	Si richiede un accesso in scrittura, di un file che viene troncato se esiste oppure viene creato. L'indice interno per l'accesso ai dati viene posizionato all'inizio del file.

`"w+"`
`"w+b"`
`"wb+"`
	Si richiede un accesso in lettura e scrittura, di un file che viene troncato se esiste oppure viene creato. L'indice interno per l'accesso ai dati viene posizionato all'inizio del file.

`"a"` `"ab"`	Si richiede la creazione o il troncamento di un file, con accesso in aggiunta. L'indice interno per l'accesso ai dati viene posizionato alla fine del file, prima di ogni operazione di scrittura.

`"a+"`
`"a+b"`
`"ab+"`
	Si richiede la creazione o il troncamento di un file, con accesso in lettura e scrittura. L'indice interno per l'accesso ai dati viene posizionato alla fine del file, prima di ogni operazione di scrittura.

VALORE RESTITUITO

Se l'operazione si conclude con successo, viene restituito il puntatore a ciò che rappresenta il flusso di file aperto. Se però si ottiene un puntatore nullo ('**`NULL`**'), si è verificato un errore che può essere interpretato dal contenuto della variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EINVAL`	È stato fornito un argomento non valido.
`EPERM`	Operazione non consentita.
`EACCES`	Accesso non consentito.
`ENOENT`	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
`EROFS`	Avendo richiesto un accesso in scrittura, si ottiene che il file system che lo contiene consente soltanto un accesso in lettura.
`ENOTDIR`	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
`ENFILE`	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
`EMFILE`	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/FILE.c`' [101.9.1]

'`lib/stdio/fopen.c`' [101.9.11]

'`lib/stdio/freopen.c`' [101.9.16]

VEDERE ANCHE

open(2) [93.28], *fclose(3)* [94.27], *stdio(3)* [94.103].

94.36 os16: fprintf(3)

Vedere *printf(3)* [94.78].

94.37 os16: fputc(3)

NOME

'**`fputc`**', '**`putc`**', '**`putchar`**' - emissione di un carattere attraverso un flusso di file

SINTASSI

```
#include <stdio.h>
int fputc   (int c, FILE *fp);
int putc    (int c, FILE *fp);
int putchar (int c);
```

DESCRIZIONE

Le funzioni ***fputc()*** e ***putc()*** sono equivalenti e scrivono il carattere ***c*** nel flusso di file rappresentato da ***fp***. La funzione ***putchar()*** esegue la scrittura di un carattere, ma nello standard output.

VALORE RESTITUITO

In caso di successo, il carattere scritto viene restituito in forma di intero positivo (il carattere viene inteso inizialmente senza segno, quindi viene trasformato in un intero, il quale rappresenta così un valore positivo). Se la scrittura non può avere luogo, la funzione restituisce '**`EOF`**', corrispondente a un valore negativo.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file associato al flusso presso cui scrivere un carattere, non è valido.
`EINVAL`	Il descrittore di file associato al flusso presso cui scrivere un carattere, non consente un accesso in scrittura.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/fputc.c`' [101.9.13]

VEDERE ANCHE

fputs(3) [94.38], *puts(3)* [94.38].

94.38 os16: fputs(3)

«

NOME

'**`fputs`**', '**`puts`**' - scrittura di una stringa attraverso un flusso di file

SINTASSI

```
#include <stdio.h>
int fputs (const char *restrict string, FILE *restrict fp);
int puts  (const char *string);
```

DESCRIZIONE

La funzione ***fputs()*** scrive una stringa nel flusso di file ***fp***, ma senza il carattere nullo di terminazione; la funzione ***puts()*** scrive una stringa, aggiungendo anche il codice di terminazione '**`\n`**', attraverso lo standard output.

VALORE RESTITUITO

Valore	Significato
≥ 0	Rappresenta il successo dell'operazione.
'**`EOF`**'	Indica il verificarsi di un errore, il quale può essere interpretato eventualmente leggendo la variabile ***errno***.

ERRORI

La variabile ***errno*** potrebbe risultare aggiornata nel caso le funzioni restituiscano '**`NULL`**'. Ma per saperlo, occorre azzerare la variabile ***errno*** prima della chiamata di queste.

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file associato al flusso verso cui si deve scrivere, non è valido.
`EINVAL`	Il descrittore di file associato al flusso verso cui si deve scrivere, non consente un accesso in scrittura.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/fputs.c`' [101.9.14]

'`lib/stdio/puts.c`' [101.9.28]

VEDERE ANCHE

fputc(3) [94.37], *putc(3)* [94.37], *putchar(3)* [94.37].

94.39 os16: fread(3)

«

NOME

'**fread**' - lettura di dati da un flusso di file

SINTASSI

```
#include <stdio.h>
size_t fread (void *restrict buffer, size_t size,
              size_t nmemb, FILE *restrict fp);
```

DESCRIZIONE

La funzione ***fread()*** legge ***size***×***nmemb*** byte dal flusso di file ***fp***, trascrivendoli in memoria a partire dall'indirizzo a cui punta ***buffer***.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letta, diviso la dimensione del blocco ***nmemb*** (byte/***nmemb***). Se il valore ottenuto è inferiore a quello richiesto, occorre verificare eventualmente se ciò deriva dalla conclusione del file o da un errore, con l'aiuto di *feof(3)* [94.28] e di *ferror(3)* [94.29].

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/fread.c`' [101.9.15]

VEDERE ANCHE

read(2) [93.29], *write(2)* [93.44], *feof(3)* [94.28], *ferror(3)* [94.29], *fwrite(3)* [94.48].

94.40 os16: free(3)

«

Vedere *malloc(3)* [94.66].

94.41 os16: freopen(3)

«

Vedere *fopen(3)* [94.35].

94.42 os16: fscanf(3)

Vedere *scanf(3)* [94.90].

94.43 os16: fseek(3)

NOME

'**fseek**', '**fseeko**' - riposizionamento dell'indice di accesso di un flusso di file

SINTASSI

```
#include <stdio.h>
int fseek  (FILE *fp, long int offset, int whence);
int fseeko (FILE *fp, off_t offset, int whence);
```

DESCRIZIONE

Le funzioni ***fseek()*** e ***fseeko()*** cambiano l'indice della posizione interna a un flusso di file, specificato dal parametro ***fp***. L'indice viene collocato secondo lo scostamento rappresentato da ***offset***, rispetto al riferimento costituito dal parametro ***whence***. Il parametro ***whence*** può assumere solo tre valori, come descritto nello schema successivo.

Valore di ***whence***	Significato
`SEEK_SET`	lo scostamento si riferisce all'inizio del file.
`SEEK_CUR`	lo scostamento si riferisce alla posizione che ha già l'indice interno al file.
`SEEK_END`	lo scostamento si riferisce alla fine del file.

La differenza tra le due funzioni sta solo nel tipo del parametro ***offset***, il quale, da '**long int**' passa a '**off_t**'.

VALORE RESTITUITO

Valore	Significato
0	Rappresenta il successo dell'operazione.
-1	Indica il verificarsi di un errore, il quale può essere interpretato leggendo la variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EBADF`	Il descrittore di file associato al flusso, non è valido.
`EINVAL`	Gli argomenti non sono validi, come succede se la combinazione di scostamento e riferimento non è ammissibile (per esempio uno scostamento negativo, quando il riferimento è l'inizio del file).

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/FILE.c`' [101.9.1]

'`lib/stdio/fseek.c`' [101.9.18]

VEDERE ANCHE

lseek(2) [93.24], *fgetpos(3)* [94.32], *fsetpos(3)* [94.32], *ftell(3)* [94.46], *rewind(3)* [94.88].

94.44 os16: fseeko(3)

Vedere *fseek(3)* [94.43].

94.45 os16: fsetpos(3)

Vedere *fgetpos(3)* [94.32].

94.46 os16: ftell(3)

NOME

'**`ftell`**', '**`ftello`**' - interrogazione dell'indice di accesso relativo a un flusso di file

SINTASSI

```
#include <stdio.h>
long int ftell  (FILE *fp);
off_t    ftello (FILE *fp);
```

DESCRIZIONE

Le funzioni ***ftell()*** e ***ftello()*** restituiscono il valore dell'indice interno di accesso al file specificato in forma di flusso, con il parametro ***fp***. La differenza tra le due

funzioni consiste nel tipo restituito, il quale, nel primo caso è '**`long int`**', mentre nel secondo è '**`off_t`**'. L'indice ottenuto è riferito all'inizio del file.

VALORE RESTITUITO

Valore	Significato
≥ 0	Rappresenta l'indice interno al file.
-1	Indica il verificarsi di un errore, il quale può essere interpretato leggendo la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file associato al flusso, non è valido.
`EINVAL`	Il flusso di file specificato non è valido.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/FILE.c`' [101.9.1]

'`lib/stdio/ftell.c`' [101.9.21]

'`lib/stdio/ftello.c`' [101.9.22]

VEDERE ANCHE

lseek(2) [93.24], *fgetpos(3)* [94.32], *fsetpos(3)* [94.32], *ftell(3)* [94.43], *rewind(3)* [94.88].

94.47 os16: ftello(3)

Vedere *ftell(3)* [94.46].

94.48 os16: fwrite(3)

NOME

'**`fwrite`**' - scrittura attraverso un flusso di file

SINTASSI

```
#include <stdio.h>
size_t fwrite (const void *restrict buffer, size_t size,
               size_t nmemb, FILE *restrict fp);
```

DESCRIZIONE

La funzione ***fwrite()*** scrive ***size***×***nmemb*** byte nel flusso di file ***fp***, traendoli dalla memoria, a partire dall'indirizzo a cui punta ***buffer***.

VALORE RESTITUITO

La funzione restituisce la quantità di byte scritta, diviso la dimensione del blocco rappresentato da ***nmemb*** (byte/***nmemb***). Se il valore ottenuto è inferiore a quello richiesto, si tratta presumibilmente di un errore, ma per accertarsene conviene usare *ferror(3)* [94.29].

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/fwrite.c`' [101.9.23]

VEDERE ANCHE

read(2) [93.29], *write(2)* [93.44], *feof(3)* [94.28], *ferror(3)* [94.29], *fread(3)* [94.39].

94.49 os16: getc(3)

«

Vedere *fgetc(3)* [94.31].

94.50 os16: getchar(3)

«

Vedere *fgetc(3)* [94.31].

94.51 os16: getenv(3)

«

NOME

'**`getenv`**' - lettura del valore di una variabile di ambiente

SINTASSI

```
#include <stdlib.h>
char *getenv (const char *name);
```

DESCRIZIONE

La funzione ***getenv()*** richiede come argomento una stringa contenente il nome di una variabile di ambiente, per poter restituire la stringa che rappresenta il contenuto di tale variabile.

VALORE RESTITUITO

Il puntatore alla stringa con il contenuto della variabile di ambiente richiesta, oppure il puntatore nullo (‘**NULL**’), se la variabile in questione non esiste.

FILE SORGENTI

‘lib/stdlib.h’ [101.10]

‘applic/crt0.s’ [102.1.9]

‘lib/stdlib/environment.c’ [101.10.9]

‘lib/stdlib/getenv.c’ [101.10.11]

VEDERE ANCHE

environ(7) [97.1], *putenv(3)* [94.82], *setenv(3)* [94.94], *unsetenv(3)* [94.94].

94.52 os16: getopt(3)

NOME

‘**getopt**’ - scansione delle opzioni della riga di comando

SINTASSI

```
#include <unistd.h>
extern *char optarg;
extern int   optind;
extern int   opterr;
extern int   optopt;
int getopt (int argc, char *const argv[],
            const char *optstring);
```

DESCRIZIONE

La funzione ***getopt()*** riceve, come primi due argomenti, gli stessi parametri ***argc*** e ***argv[]***, che sono già della funzione ***main()*** del programma in cui ***getopt()*** si usa. In altri termini, ***getopt()*** deve conoscere la quantità degli argomenti usati per l’avvio del programma e deve poterli scandire. L’ultimo argomento di ***getopt()*** è una stringa contenente l’elenco delle lettere delle opzioni che ci si attende di trovare nella scansione delle stringhe dell’array ***argv[]***, con altre sigle eventuali per sapere se tali opzioni sono singole o si attendono un proprio argomento.

Per poter usare la funzione ***getopt()*** proficuamente, è necessario che la sintassi di utilizzo del programma del quale si vuole scandire la riga di comando, sia uniforme con l’uso comune:

programma [−***x*** [***argomento***]]... [***argomento***]...

Pertanto, dopo il nome del programma possono esserci delle opzioni, riconoscibili perché composte da una sola lettera, preceduta da un trattino. Tali opzioni potrebbero richiedere un proprio argomento. Dopo le opzioni e i relativi argomenti, ci possono essere altri argomenti, al di fuori della competenza di ***getopt()***. Vale anche la considerazione che più opzioni, prive di argomento, possono essere unite assieme in un'unica parola, con un solo trattino iniziale.

La funzione ***getopt()*** si avvale di variabili pubbliche, di cui occorre conoscere lo scopo.

La variabile ***optind*** viene usata da ***getopt()*** come indice per scandire l'array ***argv[]***. Quando con gli utilizzi successivi di ***optarg()*** si determina che è stata completata la scansione delle opzioni (in quanto ***optarg()*** restituisce il valore −1), la variabile ***optind*** diventa utile per conoscere qual è il prossimo elemento di ***argv[]*** da prendere in considerazione, trattandosi del primo argomento della riga di comando che non è un'opzione.

La variabile ***opterr*** serve per configurare il comportamento di ***getopt()***. Questa variabile contiene inizialmente il valore 1. Quando ***getopt()*** incontra un'opzione per la quale si richiede un argomento, il quale risulta però mancante, se la variabile ***opterr*** risulta avere un valore diverso da zero, visualizza un messaggio di errore attraverso lo standard error. Pertanto, per evitare tale visualizzazione, è sufficiente assegnare preventivamente il valore zero alla variabile ***opterr***.

Quando un'opzione individuata da ***getopt()*** risulta errata per qualche ragione (perché non prevista o perché si attende un argomento che invece non c'è), la variabile ***optopt*** riceve il valore (tradotto da carattere senza segno a intero) della lettera corrispondente a quell'opzione. Pertanto, in tal modo è possibile conoscere cosa ha provocato il problema.

Il puntatore ***optarg*** viene modificato quando ***getopt()*** incontra un'opzione che chiede un argomento. In tal caso, ***optarg*** viene modificato in modo da puntare alla stringa che rappresenta tale argomento.

La compilazione della stringa corrispondente a ***optstring*** deve avvenire secondo una sintassi precisa:

[:] [***x*** [:]]...

La stringa ***optstring*** può iniziare con un simbolo di due punti, quindi seguono le lettere che rappresentano le opzioni possibili, tenendo conto che quelle per cui

si attende un argomento devono anche essere seguite da due punti. Per esempio, '**ab:cd:**' significa che ci può essere un'opzione '**-a**', un'opzione '**-b**' seguita da un argomento, un'opzione '**-c**' e un'opzione '**-d**' seguita da un argomento.

Per comprendere l'uso della funzione ***getopt()*** si propone una versione ultraridotta di *kill(1)* [92.10], dove si ammette solo l'invio dei segnali SIGTERM e SIGQUIT.

```
#include <sys/os16.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <libgen.h>
//----------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
  int            signal = SIGTERM;
  int            pid;
  int            a;               // Index inside arguments.
  int            opt;
  extern char *optarg;
  extern int    optopt;
  //
  while ((opt = getopt (argc, argv, ":ls:")) != -1)
    {
      switch (opt)
        {
          case 'l':
              printf ("TERM ");
              printf ("KILL ");
              printf ("\n");
              return (0);
              break;
```

```
            case 's':
                if (strcmp (optarg, "KILL") == 0)
                  {
                    signal = SIGKILL;
                  }
                else if (strcmp (optarg, "TERM") == 0)
                  {
                    signal = SIGTERM;
                  }
                break;
            case '?':
                fprintf (stderr, "Unknown option -%c.\n",
                         optopt);
                return (1);
                break;
            case ':':
                fprintf (stderr,
                         "Missing argument for option "
                         "-%c\n",
                         optopt);
                return (1);
                break;
            default:
                fprintf (stderr,
                         "Getopt problem: unknown option "
                         "%c\n", opt);
                return (1);
          }
      }
    //
    // Scan other command line arguments.
    //
    for (a = optind; a < argc; a++)
      {
        pid = atoi (argv[a]);
        if (pid > 0)
          {
            if (kill (pid, signal) < 0)
              {
```

```
                perror (argv[a]);
              }
          }
      }
    return (0);
}
```

Come si vede nell'esempio, la funzione ***getopt()*** viene chiamata sempre nello stesso modo, all'interno di un ciclo iterativo.

Alla prima chiamata della funzione, questa esamina il primo argomento della riga di comando, verificando se si tratta di un'opzione o meno. Se si tratta di un'opzione, benché possa essere errata per qualche ragione, restituisce un carattere (convertito a intero), il quale può corrispondere alla lettera dell'opzione se questa è valida, oppure a un simbolo differente in caso di problemi. Nelle chiamate successive, ***getopt()*** considera di volta in volta gli argomenti successivi della riga di comando, fino a quando si accorge che non ci sono più opzioni e restituisce semplicemente il valore -1.

Durante la scansione delle opzioni, se ***getopt()*** restituisce il carattere '**`?`**', significa che ha incontrato un'opzione errata: potrebbe trattarsi di un'opzione non prevista, oppure di un'opzione che attende un argomento che non c'è. Tuttavia, la stringa ***optstring*** potrebbe iniziare opportunamente con il simbolo di due punti, così come si vede nell'esempio. In tal caso, se ***getopt()*** incontra un'opzione errata in quanto mancante di un'opzione necessaria, invece di restituire '**`?`**', restituisce '**`:`**', così da poter distinguere il tipo di errore.

È il caso di osservare che le chiamate successive di ***getopt()*** fanno progredire la scansione della riga di comando e generalmente non c'è bisogno di tornare indietro per ripeterla. Tuttavia, nel caso lo si volesse, basterebbe reinizializzare la variabile ***optind*** a uno (il primo argomento della riga di comando).

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/getopt.c`' [101.17.20]

94.53 os16: getpwent(3)

«

NOME

'**`getpwent`**', '**`setpwent`**', '**`endpwent`**' - accesso alle voci del file '`/etc/passwd`'

SINTASSI

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwent (void);
void           setpwent (void);
void           endpwent (void);
```

DESCRIZIONE

La funzione ***getpwent()*** restituisce il puntatore a una variabile strutturata, di tipo '**`struct passwd`**', come definito nel file '`pwd.h`', in cui si possono trovare le stesse informazioni contenute nelle voci (righe) del file '`/etc/passwd`', separate in campi. La prima volta, nella variabile struttura a cui punta la funzione si ottiene il contenuto della prima voce, ovvero del primo utente dell'elenco; nelle chiamate successive si ottengono le altre.

Si utilizza la funzione ***setpwent()*** per ripartire dalla prima voce del file '`/etc/passwd`'; si utilizza invece la funzione ***endpwent()*** per chiudere il file '`/etc/passwd`' quando non serve più.

Il tipo '**`struct passwd`**' è definito nel file '`pwd.h`' nel modo seguente:

```
struct passwd {
    char   *pw_name;
    char   *pw_passwd;
    uid_t   pw_uid;
    gid_t   pw_gid;
    char   *pw_gecos;
    char   *pw_dir;
    char   *pw_shell;
};
```

La sequenza dei campi della struttura corrisponde a quella contenuta nel file '`/etc/passwd`'.

VALORE RESTITUITO

La funzione ***getpwent()*** restituisce il puntatore a una variabile strutturata di tipo '**`struct passwd`**', se l'operazione ha avuto successo. Se la scansione del file '`/etc/passwd`' ha raggiunto il termine, oppure se si è verificato un errore, restituisce invece il valore '**`NULL`**'. Per poter distinguere tra la conclusione del file o il verificarsi di un errore, prima della chiamata della funzione occorre azzera-

re il valore della variabile ***errno***, verificando successivamente se ha acquisito un valore differente.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
`EPERM`	Operazione non consentita.
`EEXIST`	Il file da creare esiste già.
`EACCES`	Accesso non consentito.
`ENOENT`	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
`EROFS`	Avendo richiesto un accesso in scrittura, si ottiene che il file system che lo contiene consente soltanto un accesso in lettura.
`ENOTDIR`	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
`ENFILE`	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
`EMFILE`	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/pwd.h`' [101.7]

'`lib/pwd/pwent.c`' [101.7.1]

VEDERE ANCHE

getpwnam(3) [94.54], *getpwuid(3)* [94.54], *passwd(5)* [96.3].

94.54 os16: getpwnam(3)

«

NOME

'**`getpwnam`**', '**`getpwuid`**' - selezione di una voce dal file '`/etc/passwd`'

SINTASSI

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwnam (const char *name);
struct passwd *getpwuid (uid_t uid);
```

DESCRIZIONE

La funzione ***getpwnam()*** restituisce il puntatore a una variabile strutturata, di tipo '**struct passwd**', come definito nel file 'pwd.h', contenente le informazioni sull'utenza specificata per nome, dal '/etc/passwd'. La funzione ***getpwuid()*** si comporta in modo analogo, individuando però l'utenza da selezionare in base al numero UID.

Il tipo '**struct passwd**' è definito nel file 'pwd.h' nel modo seguente:

```
struct passwd {
    char  *pw_name;
    char  *pw_passwd;
    uid_t  pw_uid;
    gid_t  pw_gid;
    char  *pw_gecos;
    char  *pw_dir;
    char  *pw_shell;
};
```

La sequenza dei campi della struttura corrisponde a quella contenuta nel file '/etc/passwd'.

VALORE RESTITUITO

Le funzioni ***getpwnam()*** e ***getpwuid()*** restituiscono il puntatore a una variabile strutturata di tipo '**struct passwd**', se l'operazione ha avuto successo. Se il nome o il numero dell'utente non si trovano nel file '/etc/passwd', oppure se si presenta un errore, il valore restituito è '**NULL**'. Per poter distinguere tra una voce non trovata o il verificarsi di un errore di accesso al file '/etc/passwd', prima della chiamata della funzione occorre azzerare il valore della variabile ***errno***, verificando successivamente se ha acquisito un valore differente.

ERRORI

Valore di *errno*	Significato
`EINVAL`	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
`EPERM`	Operazione non consentita.
`EEXIST`	Il file da creare esiste già.
`EACCES`	Accesso non consentito.
`ENOENT`	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
`EROFS`	Avendo richiesto un accesso in scrittura, si ottiene che il file system che lo contiene consente soltanto un accesso in lettura.
`ENOTDIR`	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
`ENFILE`	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
`EMFILE`	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/pwd.h`' [101.7]

'`lib/pwd/pwent.c`' [101.7.1]

VEDERE ANCHE

getpwent(3) [94.53], *setpwent(3)* [94.53], *endpwent(3)* [94.53], *passwd(5)* [96.3].

94.55 os16: getpwuid(3)

Vedere *getpwnam(3)* [94.54].

94.56 os16: gets(3)

Vedere *fgets(3)* [94.33].

94.57 os16: heap(3)

«

NOME

'**heap_clear**', '**heap_min**' - verifica dello spazio disponibile per la pila dei dati

SINTASSI

```
#include <sys/os16.h>
void heap_clear    (void);
int  heap_min      (void);
```

DESCRIZIONE

Le funzioni ***heap_clear()*** e ***heap_min()*** servono per poter conoscere, in un certo momento, lo spazio di memoria disponibile per la pila dei dati, durante il funzionamento del processo elaborativo.

La funzione ***heap_clear()*** sovrascrive la memoria tra la fine della memoria utilizzata per le variabili non inizializzate (BSS) e la parte superiore della pila dei dati. In altri termini, sovrascrive la parte di memoria disponibile per la pila dei dati, che in quel momento non è utilizzata. Vengono scritte sequenze di bit a uno.

La funzione ***heap_min()***, da usare successivamente a ***heap_clear()***, anche più avanti nell'esecuzione del processo, scandisce questa memoria e verifica, empiricamente, il livello minimo di memoria rimasto libero per la pila, in base all'utilizzo che se ne è fatto fino a quel punto. In pratica, serve a verificare se il programma da cui ha origine il processo ha uno spazio sufficiente per la pila dei dati o se ci sia il rischio di sovrapposizione con le altre aree dei dati.

VALORE RESTITUITO

La funzione ***heap_min()*** restituisce la quantità di byte di memoria continua, presumibilmente non ancora utilizzata dalla pila dei dati, che separa la pila stessa dalle altre aree di dati.

FILE SORGENTI

'`lib/sys/os16.h`' [101.12]

'`lib/sys/os16/heap_clear.c`' [101.12.9]

'`lib/sys/os16/heap_min.c`' [101.12.10]

VEDERE ANCHE

cs(3) [94.12], *ds(3)* [94.12], *es(3)* [94.12], *ss(3)* [94.12], *bp(3)* [94.12], *sp(3)* [94.12].

94.58 os16: heap_clear(3)

Vedere *heap(3)* [94.57].

94.59 os16: heap_min(3)

Vedere *heap(3)* [94.57].

94.60 os16: input_line(3)

NOME

'`input_line`' - riga di comando

SINTASSI

```
#include <sys/os16.h>
void input_line (char *line, char *prompt, size_t size,
                 int type);
```

DESCRIZIONE

La funzione ***input_line()*** consente di inserire un'informazione da tastiera, interpretando in modo adeguato i codici usati per cancellare. Si tratta dell'unico mezzo corretto di inserimento di un dato da tastiera, per os16, il quale non dispone di una gestione completa dei terminali.

Il parametro ***line*** è il puntatore a un'area di memoria, da modificare con l'inserimento che si intende fare; questa area di memoria deve essere in grado di contenere tanti byte quanto indicato con il parametro ***size***. Il parametro ***prompt*** indica una stringa da usare come invito, a sinistra della riga da inserire. Il parametro ***type*** serve a specificare il tipo di visualizzazione sullo schermo di ciò che si inserisce. Si utilizzano della macro-variabili dichiarate nel file '`sys/os16.h`':

Macro-variabile	Descrizione
`INPUT_LINE_ECHO`	Produce un comportamento «normale», per cui ciò che viene digitato è rappresentato conformemente sullo schermo del terminale.
`INPUT_LINE_HIDDEN`	Fa sì che quanto digitato non appaia: si usa per esempio per l'inserimento di una parola d'ordine.
`INPUT_LINE_STARS`	Fa sì che per ogni carattere digitato appaia sullo schermo un asterisco: si usa per esempio per l'inserimento di una parola d'ordine, quando si vuole agevolare l'utente.

La funzione conclude il suo funzionamento quando si preme [*Invio*].

VALORE RESTITUITO

La funzione non restituisce alcunché, ma ciò che viene digitato è disponibile nella memoria tampone rappresentata dal puntatore ***line***, da intendere come stringa terminata correttamente.

FILE SORGENTI

'`lib/sys/os16.h`' [101.12]

'`lib/sys/os16/input_line.c`' [101.12.11]

VEDERE ANCHE

shell(1) [92.19].

login(1) [92.12].

94.61 os16: isatty(3)

«

NOME

'**`isatty`**' - verifica che un certo descrittore di file si riferisca a un terminale

SINTASSI

```
#include <unistd.h>
int isatty (int fdn);
```

DESCRIZIONE

La funzione ***isatty()*** verifica se il descrittore di file specificato con il parametro ***fdn*** si riferisce a un dispositivo di terminale.

VALORE RESTITUITO

Valore	Significato
1	Si tratta effettivamente del file di dispositivo di un terminale.
0	Il descrittore di file non riguarda un terminale, oppure si è verificato un errore; in ogni caso la variabile ***errno*** viene aggiornata.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il descrittore di file non si riferisce a un terminale.
`EBADF`	Il descrittore di file indicato non è valido.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/isatty.c`' [101.17.25]

VEDERE ANCHE

stat(2) [93.36], *ttyname(3)* [94.124].

94.62 os16: labs(3)

Vedere *abs(3)* [94.3].

94.63 os16: ldiv(3)

Vedere *div(3)* [94.15].

94.64 os16: major(3)

Vedere *makedev(3)* [94.65].

94.65 os16: makedev(3)

NOME

'**`makedev`**', '**`major`**', '**`minor`**' - gestione dei numeri di dispositivo

SINTASSI

```
#include <sys/types.h>
dev_t makedev (int major, int minor);
int   major   (dev_t device);
int   minor   (dev_t device);
```

DESCRIZIONE

La funzione ***makedev()*** restituisce il numero di dispositivo complessivo, partendo dal numero primario (***major***) e dal numero secondario (***minor***), presi separatamente.

Le funzioni ***major()*** e ***minor()***, rispettivamente, restituiscono il numero primario o il numero secondario, partendo da un numero di dispositivo completo.

Si tratta di funzioni non previste dallo standard, ma ugualmente diffuse.

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/sys/types/makedev.c`' [101.14.2]

'`lib/sys/types/major.c`' [101.14.1]

'`lib/sys/types/minor.c`' [101.14.3]

94.66 os16: malloc(3)

«

NOME

'**`malloc`**', '**`free`**', '**`realloc`**' - allocazione e rilascio dinamico di memoria

SINTASSI

```
#include <stdlib.h>
void *malloc (size_t size);
void free (void *address);
void *realloc (void *address, size_t size);
```

DESCRIZIONE

Le funzioni '...**`alloc()`**' e *free()* consentono di allocare, riallocare e liberare delle aree di memoria, in modo dinamico.

La funzione ***malloc()*** (*memory allocation*) si usa per richiedere l'allocazione di una dimensione di almeno ***size*** byte di memoria. Se l'allocazione avviene con successo, la funzione restituisce il puntatore generico di tale area allocata.

Quando un'area di memoria allocata precedentemente non serve più, va liberata espressamente con l'ausilio della funzione ***free()***, la quale richiede come argomento il puntatore generico all'inizio di tale area. Naturalmente, si può liberare la memoria una volta sola e un'area di memoria liberata non può più essere raggiunta.

Quando un'area di memoria già allocata richiede una modifica nella sua estensione, si può usare la funzione ***realloc()***, la quale necessita di conoscere il puntatore precedente e la nuova estensione. La funzione restituisce un nuovo puntatore, il quale potrebbe eventualmente, ma non necessariamente, coincidere con quello dell'area originale.

Se le funzioni ***malloc()*** e ***realloc()*** falliscono nel loro intento, restituiscono un puntatore nullo.

VALORE RESTITUITO

Le funzioni ***malloc()*** e ***realloc()*** restituiscono il puntatore generico all'area di memoria allocata; se falliscono, restituiscono invece un puntatore nullo.

ERRORI

Valore di ***errno***	Significato
`ENOMEM`	Memoria insufficiente.

DIFETTI

L'allocazione dinamica di memoria, della libreria di os16, utilizza un metodo rudimentale, basato su un array statico che viene allocato completamente se nella compilazione si utilizzano queste funzioni. Questo array, denominato ***_alloc_memory[]***, viene utilizzato come area per l'allocazione della memoria, con l'ausilio di altre due variabili allo scopo di tenere traccia della mappa di allocazione. In pratica, la memoria che si può gestire in questo modo è molto poca, ma soprattutto, i processi che ne fanno uso, in realtà, la allocano subito tutta.

FILE SORGENTI

'`lib/limits.h`' [101.1.8]

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/alloc.c`' [101.10.4]

94.67 os16: memccpy(3)

NOME

'**`memccpy`**' - copia di un'area di memoria

SINTASSI

```
#include <string.h>
void *memccpy (void *restrict dst,
               const void *restrict org,
               int c, size_t n);
```

DESCRIZIONE

La funzione ***memccpy()*** copia al massimo ***n*** byte a partire dall'area di memoria a cui punta ***org***, verso l'area che inizia da ***dst***, fermandosi se si incontra il carattere

c, il quale viene copiato regolarmente, fermo restando il limite massimo di ***n*** byte.

Le due aree di memoria, origine e destinazione, non devono sovrapporsi.

VALORE RESTITUITO

Nel caso in cui la copia sia avvenuta con successo, fino a incontrare il carattere ***c***, la funzione restituisce il puntatore al carattere successivo a ***c***, nell'area di memoria di destinazione. Se invece tale carattere non viene trovato nei primi ***n*** byte, restituisce il puntatore nullo '**`NULL`**'. La variabile ***errno*** non viene modificata.

FILE SORGENTI

'`lib/string.h`' [101.11]

'`lib/string/memccpy.c`' [101.11.1]

VEDERE ANCHE

memcpy(3) [94.70], *memmove(3)* [94.71], *strcpy(3)* [94.108], *strncpy(3)* [94.108].

94.68 os16: memchr(3)

«

NOME

'**`memchr`**' - scansione della memoria alla ricerca di un carattere

SINTASSI

```
#include <string.h>
void *memchr (const void *memory, int c, size_t n);
```

DESCRIZIONE

La funzione ***memchr()*** scandisce l'area di memoria a cui punta ***memory***, fino a un massimo di ***n*** byte, alla ricerca del carattere ***c***.

VALORE RESTITUITO

Se la funzione trova il carattere, restituisce il puntatore al carattere trovato, altrimenti restituisce il puntatore nullo '**`NULL`**'.

FILE SORGENTI

'`lib/string.h`' [101.11]

'`lib/string/memchr.c`' [101.11.2]

VEDERE ANCHE

strchr(3) [94.105], *strrchr(3)* [94.105], *strpbrk(3)* [94.116].

94.69 os16: memcmp(3)

NOME

'**memcmp**' - confronto di due aree di memoria

SINTASSI

```
#include <string.h>
int memcmp (const void *memory1, const void *memory2,
            size_t n);
```

DESCRIZIONE

La funzione ***memcmp()*** confronta i primi ***n*** byte di memoria delle aree che partono, rispettivamente, da ***memory1*** e da ***memory2***.

VALORE RESTITUITO

Valore	Esito del confronto.
−1	***memory1 < memory2***
0	***memory1 == memory2***
+1	***memory1 > memory2***

FILE SORGENTI

'lib/string.h' [101.11]

'lib/string/memcmp.c' [101.11.3]

VEDERE ANCHE

strcmp(3) [94.106], *strncmp(3)* [94.106].

94.70 os16: memcpy(3)

NOME

'**memcpy**' - copia di un'area di memoria

SINTASSI

```
#include <string.h>
void *memcpy (void *restrict dst, const void *restrict org,
              size_t n);
```

DESCRIZIONE

La funzione ***memcpy()*** copia al massimo ***n*** byte a partire dall'area di memoria a cui punta ***org***, verso l'area che inizia da ***dst***.

Le due aree di memoria, origine e destinazione, non devono sovrapporsi.

VALORE RESTITUITO

La funzione restituisce ***dst***.

FILE SORGENTI

'`lib/string.h`' [101.11]

'`lib/string/memcpy.c`' [101.11.4]

VEDERE ANCHE

memccpy(3) [94.67], *memmove(3)* [94.71], *strcpy(3)* [94.108], *strncpy(3)* [94.108].

94.71 os16: memmove(3)

«

NOME

'**`memmove`**' - copia di un'area di memoria

SINTASSI

```
#include <string.h>
void *memmove (void *dst, const void *org, size_t n);
```

DESCRIZIONE

La funzione ***memmove()*** copia al massimo ***n*** byte a partire dall'area di memoria a cui punta ***org***, verso l'area che inizia da ***dst***. A differenza di quanto fa ***memcpy()***, la funzione ***memmove()*** esegue la copia correttamente anche se le due aree di memoria sono sovrapposte.

VALORE RESTITUITO

La funzione restituisce ***dst***.

FILE SORGENTI

'`lib/string.h`' [101.11]

'`lib/string/memmove.c`' [101.11.5]

VEDERE ANCHE

memccpy(3) [94.67], *memcpy(3)* [94.70], *strcpy(3)* [94.108], *strncpy(3)* [94.108].

94.72 os16: memset(3)

NOME

'**memset**' - scrittura della memoria con un byte sempre uguale

SINTASSI

```
#include <string.h>
void *memset (void *memory, int c, size_t n);
```

DESCRIZIONE

La funzione ***memset()*** scrive ***n*** byte, contenenti il valore di ***c***, ridotto a un carattere, a partire dal ciò a cui punta ***memory***.

FILE SORGENTI

'lib/string.h' [101.11]

'lib/string/memset.c' [101.11.6]

VEDERE ANCHE

memcpy(3) [94.70].

94.73 os16: minor(3)

Vedere *makedev(3)* [94.65].

94.74 os16: namep(3)

NOME

'**namep**' - ricerca del percorso di un programma utilizzando la variabile di ambiente ***PATH***

SINTASSI

```
#include <sys/os16.h>
int namep (const char *name, char *path, size_t size);
```

DESCRIZIONE

La funzione ***namep()*** trova il percorso di un programma, tenendo conto delle informazioni contenute nella variabile di ambiente ***PATH***.

Il parametro ***name*** rappresenta una stringa con il nome del comando da cercare nel file system; il parametro ***path*** deve essere il puntatore di un'area di memoria, da sovrascrivere con il percorso assoluto del programma da avviare, una volta trovato, con l'accortezza di far sì che risulti una stringa terminata correttamente; il parametro ***size*** specifica la dimensione massima che può avere la stringa ***path***.

Questa funzione viene utilizzata in particolare da ***execvp()***.

VALORE RESTITUITO

Valore	Significato
0	Operazione riuscita.
−1	Operazione fallita. Va verificato l'errore indicato dalla variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Uno degli argomenti non è valido.
`ENOENT`	La variabile di ambiente ***PATH*** non è dichiarata, oppure il comando richiesto non si trova nei percorsi previsti.
`ENAMETOOLONG`	Il percorso per l'avvio del programma è troppo lungo.

FILE SORGENTI

'`lib/sys/os16.h`' [101.12]

'`lib/sys/os16/namep.c`' [101.12.13]

VEDERE ANCHE

shell(1) [92.19], *execvp(3)* [94.20], *execlp(3)* [94.20].

94.75 os16: offsetof(3)

«

NOME

'**`offsetof`**' - posizione di un membro di una struttura, dall'inizio della stessa

SINTASSI

```
#include <stddef.h>
size_t offsetof (type, member);
```

DESCRIZIONE

La macro-istruzione ***offsetof()*** consente di determinare la collocazione relativa di un membro di una variabile strutturata, restituendo la quantità di byte che la struttura occupa prima dell'inizio del membro richiesto. Per ottenere questo risultato, il primo argomento deve essere il nome del tipo del membro cercato, mentre il secondo argomento deve essere il nome del membro stesso.

VALORE RESTITUITO

La macro-istruzione restituisce lo scostamento del membro specificato, rispetto all'inizio della struttura a cui appartiene, espresso in byte.

FILE SORGENTI

'`lib/stddef.h`' [101.1.14]

94.76 os16: opendir(3)

«

NOME

'**`opendir`**' - apertura di una directory

SINTASSI

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *name);
```

DESCRIZIONE

La funzione ***opendir()*** apre la directory rappresentata da ***name***, posizionando l'indice interno per le operazioni di accesso alla prima voce della directory stessa.

VALORE RESTITUITO

La funzione restituisce il puntatore al flusso aperto; in caso di errore, restituisce '**`NULL`**' e aggiorna la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il valore di ***name*** non è valido.
`EMFILE`	Troppi file aperti dal processo.
`ENFILE`	Troppi file aperti complessivamente nel sistema.
`ENOTDIR`	Il percorso indicato in ***name*** non corrisponde a una directory.

NOTE

La funzione ***opendir()*** attiva il bit *close-on-exec*, rappresentato dalla macro-variabile ***FD_CLOEXEC***, per il descrittore del file che rappresenta la directory. Ciò serve a garantire che la directory venga chiusa quando si utilizzano le funzioni '**exec...()**'.

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/dirent.h`' [101.2]

'`lib/dirent/DIR.c`' [101.2.1]

'`lib/dirent/opendir.c`' [101.2.3]

VEDERE ANCHE

open(2) [93.28], *closedir(3)* [94.10], *readdir(3)* [94.86], *rewinddir(3)* [94.89].

94.77 os16: perror(3)

«

NOME

'**perror**' - emissione di un messaggio di errore di sistema

SINTASSI

```
#include <stdio.h>
void perror (const char *string);
```

DESCRIZIONE

La funzione ***perror()*** legge il valore della variabile ***errno*** e, se questo è diverso da zero, emette attraverso lo standard output la stringa fornita come argomento,

ammesso che non si tratti del puntatore nullo, quindi continua con l'emissione della descrizione dell'errore.

La funzione ***perror()*** di os16, nell'emettere il testo dell'errore, mostra anche il nome del file sorgente e il numero della riga in cui si è verificato. Ma questi dati sono validi soltanto se l'annotazione dell'errore è avvenuta, a suo tempo, con l'ausilio della funzione *errset(3)* [94.18], la quale non è prevista dagli standard.

FILE SORGENTI

'`lib/errno.h`' [101.3]

'`lib/stdio.h`' [101.9]

'`lib/stdio/perror.c`' [101.9.26]

VEDERE ANCHE

errno(3) [94.18], *strerror(3)* [94.111].

94.78 os16: printf(3)

NOME

'**`printf`**', '**`fprintf`**', '**`sprintf`**', '**`snprintf`**' - composizione dei dati per la visualizzazione

SINTASSI

```
#include <stdio.h>
int printf   (char *restrict format, ...);
int fprintf  (FILE *fp, char *restrict format, ...);
int sprintf  (char *restrict string,
              const char *restrict format, ...);
int snprintf (char *restrict string, size_t size,
              const char *restrict format, ...);
```

DESCRIZIONE

Le funzioni del gruppo '...**`printf()`**' hanno in comune lo scopo di comporre dei dati in forma di stringa, generalmente per la visualizzazione, o comunque per la fruizione a livello umano.

I dati in ingresso possono essere vari e si collocano come argomenti finali, di tipo e quantità non noti nel prototipo delle funzioni. Per quantificare e qualificare

questi argomenti aggiuntivi, la stringa a cui punta il parametro ***format***, deve contenere degli ***specificatori di conversione***, oltre eventualmente ad altri caratteri. Pertanto, queste funzioni, prendono la stringa a cui punta ***format***, la interpretano e determinano quali argomenti variabili sono presenti, quindi producono un'altra stringa, composta dalla stringa precedente, sostituendo gli specificatori di conversione con i dati a cui questi si riferiscono, secondo una forma di conversione definita dagli specificatori stessi. Si osservi l'esempio seguente:

```
printf ("Valore: %x %i %o\n", 123, 124, 125);
```

In questo modo si ottiene la visualizzazione, attraverso lo standard output, della stringa '**`Valore: 7b 124 175`**'. Infatti: '**`%x`**' è uno specificatore di conversione che richiede di interpretare il proprio parametro (in questo caso il primo) come intero normale e di rappresentarlo in esadecimale; '**`%i`**' legge un numero intero normale e lo rappresenta nella forma decimale consueta; '**`%o`**' legge un intero e lo mostra in ottale.

La funzione ***printf()*** emette il risultato della composizione attraverso lo standard output; la funzione ***fprintf()*** lo fa attraverso il flusso di file ***fp***; le funzioni ***sprintf()*** e ***snprintf()*** si limitano a scrivere il risultato a partire da ciò a cui punta ***string***, con la particolarità di ***snprintf()*** che si dà comunque un limite da non superare, per evitare che la scrittura vada a sovrascrivere altri dati in memoria.

Gli specificatori di conversione devono rispettare la sintassi seguente per la libreria di os16:

`%`[***simbolo***] [***n_ampiezza***] [***.n_precisione***] [`hh`|`h`|`l`|`j`|`z`|`t`] ***tipo***

La prima cosa da individuare in uno specificatore di conversione è il tipo di argomento che viene interpretato e, di conseguenza, il genere di rappresentazione che se ne vuole produrre. Il tipo viene espresso da una lettera alfabetica, alla fine dello specificatore di conversione.

Simbolo	Tipo di argomento	Conversione applicata
`%…d` `%…i`	`int`	Numero intero con segno da rappresentare in base dieci.
`%…u`	`unsigned int`	Numero intero senza segno da rappresentare in base dieci.
`%…o`	`unsigned int`	Numero intero senza segno da rappresentare in ottale (senza lo zero iniziale che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
`%…x` `%…X`	`unsigned int`	Numero intero senza segno da rappresentare in esadecimale (senza il prefisso ‘**`0x`**’ o ‘**`0X`**’ che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
`%…c`	`int`	Un carattere singolo, dopo la conversione in ‘**`unsigned char`**’.
`%…s`	`char *`	Una stringa.
`%%`		Questo specificatore si limita a produrre un carattere di percentuale (‘**`%`**’) che altrimenti non sarebbe rappresentabile.

Nel modello sintattico che descrive lo specificatore di conversione, si vede che subito dopo il segno di percentuale può apparire un simbolo (*flag*).

Simbolo	Corrispondenza
`%+`... `%+0`***ampiezza***...	Il segno «+» fa sì che i numeri con segno lo mostrino anche se è positivo. Può combinarsi con lo zero.
`%0`***ampiezza***... `%+0`***ampiezza***...	Lo zero fa sì che siano inseriti degli zeri a sinistra per allineare a destra il valore, nell'ambito dell'ampiezza specificata. Può combinarsi con il segno «+».
`%`***ampiezza***... `% `***ampiezza***...	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi. È possibile esprimere esplicitamente l'intenzione di usare gli spazi mettendo proprio uno spazio, ma in generale non è richiesto. Se si mette lo spazio letteralmente, questo non è poi compatibile con lo zero, mentre le combinazioni con gli altri simboli sono ammissibili.
`%-`***ampiezza***... `%-+`***ampiezza***...	Il segno meno, usato quando la conversione prevede l'uso di una quantità fissa di caratteri con un valore che appare di norma allineato a destra, fa sì che il risultato sia allineato a sinistra. Il segno meno si può combinare il segno «+» e il cancelletto.

Subito prima della lettera che definisce il tipo di conversione, possono apparire una o due lettere che modificano la lunghezza del valore da interpretare (per lunghezza si intende qui la quantità di byte usati per rappresentarlo). Per esempio, '**`%...li`**' indica che la conversione riguarda un valore di tipo '**`long int`**'. Tra questi specificatori della lunghezza del dato in ingresso ce ne sono alcuni che indicano un rango inferiore a quello di '**`int`**', come per esempio '**`%...hhd`**' che si riferisce a un numero intero della dimensione di un '**`signed char`**'; in questi casi occorre comunque considerare che nella trasmissione degli argomenti alle funzioni interviene sempre la promozione a intero, pertanto viene letto il dato della dimensione specificata, ma viene «consumato» il risultato ottenuto dalla promozione.

<table>
<tr><th>Simbolo</th><th>Tipo</th><th>Simbolo</th><th>Tipo</th></tr>
<tr><td>%…hhd
%…hhi</td><td>signed char</td><td>%…hhu
%…hho
%…hhx |
%…hhX</td><td>unsigned char</td></tr>
<tr><td>%…hd
%…hi</td><td>short int</td><td>%…hu
%…ho
%…hx | %…hX</td><td>unsigned short int</td></tr>
<tr><td>%…ld
%…li</td><td>long int</td><td>%…lu
%…lo
%…lx | %…lX</td><td>unsigned long int</td></tr>
<tr><td>%…jd
%…ji</td><td>intmax_t</td><td>%…ju
%…jo
%…jx | %…jX</td><td>uintmax_t</td></tr>
<tr><td>%…zd
%…zi</td><td>size_t</td><td>%…zu
%…zo
%…zx | %…zX</td><td>size_t</td></tr>
<tr><td>%…td
%…ti</td><td>ptrdiff_t</td><td>%…tu
%…to
%…tx | %…tX</td><td>ptrdiff_t</td></tr>
</table>

Tra il simbolo (*flag*) e il modificatore di lunghezza può apparire un numero che

rappresenta l'ampiezza da usare nella trasformazione ed eventualmente la precisione: '***ampiezza*** [**.*precisione***]'. Per os16, la precisione si applica esclusivamente alle stringhe, la quale specifica la quantità di caratteri da considerare, troncando il resto.

VALORE RESTITUITO

Le funzioni restituiscono la quantità di caratteri utilizzati nella composizione della nuova stringa, escluso il carattere nullo di terminazione.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/FILE.c`' [101.9.1]

'`lib/stdio/fprintf.c`' [101.9.12]

'`lib/stdio/printf.c`' [101.9.27]

'`lib/stdio/sprintf.c`' [101.9.34]

'`lib/stdio/snprintf.c`' [101.9.33]

VEDERE ANCHE

vfprintf(3) [94.128], *vprintf(3)* [94.128], *vsprintf(3)* [94.128], *vsnprintf(3)* [94.128], *scanf(3)* [94.90].

94.79 os16: process_info(3)

«

NOME

'**`process_info`**' - funzione diagnostica

SINTASSI

```
#include <sys/os16.h>
void process_info (void);
```

DESCRIZIONE

Si tratta di una funzione diagnostica che non richiede argomenti e non restituisce alcunché, per visualizzare, attraverso lo standard output, lo stato dei registri della CPU, i riferimenti principali della collocazione in memoria del processo elaborativo e lo spazio ancora non utilizzato dalla pila dei dati.

Per poter dare un'informazione utile sullo spazio non ancora utilizzato dalla pila dei dati, occorre che prima di questa funzione sia stata chiamata ***heap_clear()***.

FILE SORGENTI

'lib/sys/os16.h' [101.12]

'lib/sys/os16/process_info.c' [101.12.14]

VEDERE ANCHE

cs(3) [94.12].

ds(3) [94.12].

es(3) [94.12].

ss(3) [94.12].

bp(3) [94.12].

sp(3) [94.12].

heap_clear(3) [94.57].

heap_min(3) [94.57].

94.80 os16: putc(3)

Vedere *fputc(3)* [94.37].

94.81 os16: putchar(3)

Vedere *fputc(3)* [94.37].

94.82 os16: putenv(3)

NOME

'**putenv**' - assegnamento di una variabile di ambiente

SINTASSI

```
#include <stdlib.h>
int putenv (const char *string);
```

DESCRIZIONE

La funzione ***putenv()*** assegna una variabile di ambiente. Se questa esiste già, va a rimpiazzare il valore assegnatole in precedenza, altrimenti la crea contestualmente.

La funzione richiede un solo parametro, costituito da una stringa in cui va specificato il nome della variabile e il contenuto da assegnargli, usando la forma '***nome*=*valore***'. Per esempio, '**PATH=/bin:/usr/bin**'.

VALORE RESTITUITO

Valore	Significato
0	Operazione riuscita.
−1	Operazione fallita. Va verificato l'errore indicato dalla variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Argomento non valido.
`ENOMEM`	Memoria insufficiente.

FILE SORGENTI

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/environment.c`' [101.10.9]

'`lib/stdlib/putenv.c`' [101.10.14]

VEDERE ANCHE

environ(7) [97.1], *getenv(3)* [94.51], *setenv(3)* [94.94], *unsetenv(3)* [94.94].

94.83 os16: puts(3)

Vedere *fputs(3)* [94.38].

94.84 os16: qsort(3)

NOME

'**`qsort`**' - riordino di un array

SINTASSI

```
#include <stdlib.h>
void qsort (void *base, size_t nmemb, size_t size,
            int (*compare)(const void *, const void *));
```

DESCRIZIONE

La funzione ***qsort()*** riordina un array composto da ***nmemb*** elementi da ***size*** byte ognuno. Il primo argomento, ovvero il parametro ***base***, è il puntatore all'indirizzo iniziale di questo array in memoria.

Il riordino avviene comparando i vari elementi con l'ausilio di una funzione, passata tramite il suo puntatore, la quale deve ricevere due argomenti, costituiti dai puntatori agli elementi dell'array da confrontare. Tale funzione deve restituire un valore minore di zero per un confronto in cui il suo primo argomento deve essere collocato prima del secondo; un valore pari a zero se gli argomenti sono uguali ai fini del riordino; un valore maggiore di zero se il suo primo argomento va collocato dopo il secondo nel riordino.

Segue un esempio di utilizzo della funzione ***qsort()***:

```
#include <stdio.h>
#include <stdlib.h>

int confronta (const void *a, const void *b)
{
    int x = *((int *) a);
    int y = *((int *) b);
    return x - y;
}

int main (void)
{
    int a[] = {3, 1, 5, 2};

    qsort (&a[0], 4, sizeof (int), confronta);
    printf ("%d %d %d %d\n", a[0], a[1], a[2], a[3]);

    return 0;
}
```

FILE SORGENTI

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/qsort.c`' [101.10.15]

94.85 os16: rand(3)

NOME

'**`rand`**' - generazione di numeri pseudo-casuali

SINTASSI

```
#include <stdlib.h>
int  rand  (void);
void srand (unsigned int seed);
```

DESCRIZIONE

La funzione ***rand()*** produce un numero intero casuale, sulla base di un seme, il quale può essere cambiato in ogni momento, con l'ausilio di ***srand()***. A ogni chiamata della funzione ***rand()***, il risultato ottenuto, viene utilizzato anche come seme per la chiamata successiva. Se inizialmente non viene assegnato alcun seme, il primo valore predefinito è pari a 1.

VALORE RESTITUITO

La funzione ***rand()*** restituisce un numero intero casuale, determinato sulla base del seme accumulato in precedenza.

FILE SORGENTI

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/rand.c`' [101.10.16]

94.86 os16: readdir(3)

«

NOME

'**`readdir`**' - lettura di una directory

SINTASSI

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir (DIR *dp);
```

DESCRIZIONE

La funzione ***readdir()*** legge una voce dalla directory rappresentata da ***dp*** e restituisce il puntatore a una variabile strutturata di tipo '**`struct dirent`**', contenente le informazioni tratte dalla voce letta. La variabile strutturata in questione si trova in memoria statica e viene sovrascritta con le chiamate successive della funzione ***readdir()***.

Il tipo ‘**struct dirent**’ è definito nel file di intestazione ‘dirent.h’, nel modo seguente:

```
struct dirent {
    ino_t       d_ino;
    char        d_name[NAME_MAX+1];
};
```

Il membro ***d_ino*** è il numero di inode del file il cui nome appare nel membro ***d_name***. La macro-variabile ***NAME_MAX*** è dichiarata a sua volta nel file di intestazione ‘limits.h’. La dimensione del membro ***d_name*** è tale da permettere di includere anche il valore zero di terminazione delle stringhe.

VALORE RESTITUITO

La funzione restituisce il puntatore a una variabile strutturata di tipo ‘**struct dirent**’; se la lettura ha già raggiunto la fine della directory, oppure per qualunque altro tipo di errore, la funzione restituisce ‘**NULL**’ e aggiorna eventualmente la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
EBADF	La directory rappresentata da ***dp*** non è valida.

FILE SORGENTI

‘lib/sys/types.h’ [101.14]

‘lib/dirent.h’ [101.2]

‘lib/dirent/DIR.c’ [101.2.1]

‘lib/dirent/readdir.c’ [101.2.4]

VEDERE ANCHE

read(2) [93.29], *closedir(3)* [94.10], *opendir(3)* [94.76], *rewinddir(3)* [94.89].

94.87 os16: realloc(3)

Vedere *malloc(3)* [94.66].

94.88 os16: rewind(3)

«

NOME

'**rewind**' - riposizionamento all'inizio dell'indice di accesso a un flusso di file

SINTASSI

```
#include <stdio.h>
void rewind (FILE *fp);
```

DESCRIZIONE

La funzione ***rewind()*** azzera l'indice della posizione interna del flusso di file specificato con il parametro ***fp***; inoltre azzera anche l'indicatore di errore dello stesso flusso. In pratica si ottiene la stessa cosa di:

```
(void) fseek (fp, 0L, SEEK_SET);
clearerr (fp);
```

FILE SORGENTI

'lib/stdio.h' [101.9]

'lib/stdio/FILE.c' [101.9.1]

'lib/stdio/rewind.c' [101.9.29]

VEDERE ANCHE

lseek(2) [93.24], *fgetpos(3)* [94.32], *fsetpos(3)* [94.32], *ftell(3)* [94.46], *fseek(3)* [94.43], *rewind(3)* [94.88].

94.89 os16: rewinddir(3)

«

NOME

'**rewinddir**' - riposizionamento all'inizio del riferimento per l'accesso a una directory

SINTASSI

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir (DIR *dp);
```

DESCRIZIONE

La funzione ***rewinddir()*** riposiziona i riferimenti per l'accesso alla directory indicata, in modo che la prossima lettura o scrittura avvenga dalla prima posizione.

VALORE RESTITUITO

La funzione non restituisce alcunché e non si presenta nemmeno la possibilità di segnalare errori attraverso la variabile ***errno***.

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/dirent.h`' [101.2]

'`lib/dirent/DIR.c`' [101.2.1]

'`lib/dirent/rewinddir.c`' [101.2.5]

VEDERE ANCHE

rewind(3) [94.88], *closedir(3)* [94.10], *opendir(3)* [94.76], *rewinddir(3)* [94.86].

94.90 os16: scanf(3)

NOME

'**`scanf`**', '**`fscanf`**', '**`sscanf`**' - interpretazione dell'input e conversione

SINTASSI

```
#include <stdio.h>
int scanf  (const char *restrict format, ...);
int fscanf (FILE *restrict fp,
            const char *restrict format, ...);
int sscanf (char *restrict string,
            const char *restrict format, ...);
```

DESCRIZIONE

Le funzioni del gruppo '...**`scanf()`**' hanno in comune lo scopo di interpretare dei dati, forniti in forma di stringa, convertendoli opportunamente.

I dati in ingresso sono costituiti da una sequenza di caratteri, la quale viene fornita tramite lo standard input per ***scanf()***, tramite il flusso di file ***fp*** per ***fscanf()***, oppure tramite la stringa ***string*** per ***sscanf()***. L'interpretazione dei dati in ingresso viene guidata da una stringa di formato, costituita dal parametro ***format***, per

le tre funzioni. La stringa di formato contiene degli ***specificatori di conversione***, con cui si determina il tipo degli argomenti variabili che non sono esplicitati nel prototipo delle funzioni.

Per ogni specificatore di conversione contenuto nella stringa di formato, deve esistere un argomento, successivo al parametro ***format***, costituito dal puntatore a una variabile di tipo conforme a quanto indicato dallo specificatore relativo. La conversione per quello specificatore, comporta la memorizzazione del risultato in memoria, in corrispondenza del puntatore relativo. Si osservi l'esempio seguente:

```
int valore;
…
scanf ("%i", &valore);
```

In questo modo si attende l'inserimento, attraverso lo standard input, di un numero intero, da convertire e assegnare così alla variabile ***valore***; Infatti, lo specificatore di conversione '**%i**', consente di interpretare un numero intero.

Gli specificatori di conversione devono rispettare la sintassi seguente per la libreria di os16:

```
%[*][n_ampiezza][hh|h|l|j|z|t]tipo
```

Come si può vedere, all'inizio può apparire un asterisco, il cui scopo è quello di annullare l'assegnamento del valore a una variabile. In pratica, con l'asterisco il dato corrispondente allo specificatore viene interpretato, ma poi non viene salvato.

Successivamente può apparire un numero che rappresenta l'ampiezza del dato da interpretare, in byte, il cui scopo è quello di limitare la lettura fino a un certo carattere.

Dopo può apparire una sigla, composta da una o più lettere, il cui scopo è quello di modificare la dimensione predefinita della variabile di destinazione. In altri termini, senza questo modificatore si intende che la variabile ricevente debba essere di una certa grandezza, ma con l'aggiunta del «modificatore di lunghezza» si precisa invece qualcosa di diverso. In pratica, il modificatore di lunghezza usato da queste funzioni è equivalente a quello delle funzioni di composizione dell'output.

Al termine dello specificatore di conversione appare una lettera che dichiara come deve essere interpretato il dato in ingresso e, in mancanza del modificatore di lunghezza, indica anche la dimensione della variabile ricevente.

Tipi di conversione.

Simbolo	Tipo di argomento	Conversione applicata
`%...d`	`int *`	Numero intero con segno rappresentato in base dieci.
`%...i`	`int *`	Numero intero con segno rappresentare in base dieci o in base otto, avendo come prefisso uno zero, oppure in base sedici, avendo come prefisso '**`0x`**' o '**`0X`**'.
`%...u`	`unsigned int *`	Numero intero senza segno rappresentato in base dieci.
`%...o`	`unsigned int *`	Numero intero senza segno rappresentato in ottale (con o senza lo zero iniziale).

Simbolo	Tipo di argomento	Conversione applicata
`%...x`	`unsigned int *`	Numero intero senza segno rappresentato in esadecimale (con o senza il prefisso '**`0x`**' o '**`0X`**').
`%...c`	`char *`	Interpreta un solo carattere, o più caratteri se si specifica l'ampiezza. Nella lettura contano anche gli spazi o qualunque altro carattere e non viene aggiunto il carattere nullo di terminazione.
`%...s`	`char *`	Interpreta una sequenza di caratteri che non siano spazi, aggiungendo alla fine il carattere nullo di terminazione.
`%p`	`void *`	Interpreta il valore di un puntatore che sia rappresentato nello stesso modo in cui farebbe la funzione '**`printf("%p",`** ***puntatore*****`)`**'.

Simbolo	Tipo di argomento	Conversione applicata
`%n`	`int *`	Questo specificatore non esegue alcuna conversione e si limita a memorizzare la quantità di caratteri ('**`char`**') letti fino a quel punto.
`%…[…]`	`char *`	Interpreta una stringa non vuota contenente solo i caratteri elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri si cerca anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: '**`%…[]…]`**'.
`%…[^…]`	`char *`	Interpreta una stringa non vuota contenente solo caratteri diversi da quelli elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri da escludere si vuole indicare anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: '**`%…[^]…]`**'.
`%%`		Interpreta un carattere di percentuale tra i dati in ingresso, ma senza memorizzare alcunché.

Modificatori della lunghezza del dato in uscita.

Simbolo	Tipo	Simbolo	Tipo

`%...hhd`
`%...hhi`
	`signed char *`	`%...hhu`
		`%...hho`
		`%...hhx`
		`%...hhn`
			`unsigned char *`

`%...hd`
`%...hi`
	`short int *`	`%...hu`
		`%...ho`
		`%...hx`
		`%...hn`
			`unsigned short int *`

`%...ld`
`%...li`
	`long int *`	`%...lu`
		`%...lo`
		`%...lx`
		`%...ln`
			`unsigned long int *`

Simbolo	Tipo	Simbolo	Tipo
`%…jd` `%…ji`	`intmax_t *`	`%…ju` `%…jo` `%…jx` `%…jn`	`uintmax_t *`
`%…zd` `%…zi`	`size_t *`	`%…zu` `%…zo` `%…zx` `%…zn`	`size_t *`
`%…td` `%…ti`	`ptrdiff_t *`	`%…tu` `%…to` `%…tx` `%…tn`	`ptrdiff_t *`

La stringa di conversione è composta da ***direttive***, ognuna delle quali è formata da: uno o più spazi (spazi veri e propri o caratteri di tabulazione orizzontale); un carattere diverso da '`%`' e diverso dai caratteri che rappresentano spazi, oppure uno specificatore di conversione.

[***spazi***] ***carattere*** | `%…`

Dalla sequenza di caratteri che costituisce i dati in ingresso da interpretare, vengono eliminati automaticamente gli spazi iniziali e finali (tutto ciò che si può considerare spazio, anche il codice di interruzione di riga), quando all'inizio o alla fine non ci sono corrispondenze con specificatori di conversione che possono interpretarli.

Quando la direttiva di interpretazione inizia con uno o più spazi orizzontali, significa che si vogliono ignorare gli spazi a partire dalla posizione corrente nella

lettura dei dati in ingresso; inoltre, la presenza di un carattere che non fa parte di uno specificatore di conversione indica che quello stesso carattere deve essere incontrato nell'interpretazione dei dati in ingresso, altrimenti il procedimento di lettura e valutazione si deve interrompere. Se due specificatori di conversione appaiono adiacenti, i dati in ingresso corrispondenti possono essere separati da spazi orizzontali o da spazi verticali (il codice di interruzione di riga).

VALORE RESTITUITO

Le funzioni restituiscono la quantità di elementi in ingresso interpretati e assegnati correttamente: una quantità inferiore al previsto indica pertanto un errore. Se le funzioni restituiscono il valore '**`EOF`**', si tratta di un errore, dovuto eventualmente a un problema di interpretazione del formato o a un problema di accesso al flusso di file da cui deve provenire l'input.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
`EPERM`	Operazione di accesso non consentita.
`EACCES`	Accesso non consentito.
`EBADF`	Il descrittore del file a cui si riferisce il flusso, non è valido.
`ERANGE`	Il risultato della conversione di un intero, non può essere memorizzato nel tipo di variabile a cui si riferisce lo specificatore di conversione.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/fscanf.c`' [101.9.17]

'`lib/stdio/scanf.c`' [101.9.30]

'`lib/stdio/sscanf.c`' [101.9.35]

VEDERE ANCHE

vfscanf(3) [94.129], *vscanf(3)* [94.129], *vsscanf(3)* [94.129], *printf(3)* [94.78].

94.91 os16: seg_d(3)

«

NOME

'**seg_d**', '**seg_i**' - collocazione del processo in memoria

SINTASSI

```
#include <sys/os16.h>
unsigned int seg_d (void);
unsigned int seg_i (void);
```

DESCRIZIONE

Le funzioni elencate nel quadro sintattico, sono in realtà delle macro-istruzioni, chiamanti funzioni con nomi analoghi, ma preceduti da un trattino basso (***_seg_d()*** e ***_seg_i()***), per interrogare, rispettivamente, lo stato del registro ***DS*** e ***CS***. Questi due registri indicano, rispettivamente, la collocazione dell'area dati e dell'area codice del processo in corso. Eventualmente, per conoscere l'indirizzo efficace di memoria corrispondente, occorre moltiplicare questi valori per 16.

FILE SORGENTI

'`lib/sys/os16.h`' [101.12]

'`lib/sys/os16/_seg_i.s`' [101.12.6]

'`lib/sys/os16/_seg_d.s`' [101.12.5]

VEDERE ANCHE

cs(3) [94.12].

ds(3) [94.12].

es(3) [94.12].

ss(3) [94.12].

bp(3) [94.12].

sp(3) [94.12].

94.92 os16: seg_i(3)

«

Vedere *seg_d(3)* [94.91].

94.93 os16: setbuf(3)

NOME

'**setbuf**', '**setvbuf**' - modifica della memoria tampone per i flussi di file

SINTASSI

```
#include <stdio.h>
void setbuf (FILE *restrict fp, char *restrict buffer);
int setvbuf (FILE *restrict fp, char *restrict buffer,
             int buf_mode, size_t size);
```

DESCRIZIONE

Le funzioni ***setbuf()*** e ***setvbuf()*** della libreria di os16, non fanno alcunché, perché os16 non gestisce una memoria tampone per i flussi di file.

VALORE RESTITUITO

La funzione ***setvbuf()*** restituisce, in tutti i casi, il valore zero.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/setbuf.c`' [101.9.31]

'`lib/stdio/setvbuf.c`' [101.9.32]

VEDERE ANCHE

fflush(3) [94.30].

94.94 os16: setenv(3)

NOME

'**setenv**', '**unsetenv**' - assegnamento o cancellazione di una variabile di ambiente

SINTASSI

```
#include <stdlib.h>
int setenv   (const char *name, const char *value,
              int overwrite);
int unsetenv (const char *name);
```

DESCRIZIONE

La funzione ***setenv()*** crea o assegna un valore a una variabile di ambiente. Se questa variabile esiste già, la modifica del valore assegnatole può avvenire soltanto se l'argomento corrispondente al parametro ***overwrite*** risulta essere diverso da zero; in caso contrario, la modifica non ha luogo.

La funzione ***unsetenv()*** si limita a cancellare la variabile di ambiente specificata come argomento.

VALORE RESTITUITO

Valore	Significato
0	Operazione riuscita.
−1	Operazione fallita. Va verificato l'errore indicato dalla variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Argomento non valido.
`ENOMEM`	Memoria insufficiente.

FILE SORGENTI

'`lib/stdlib.h`' [101.10]

'`applic/crt0.s`' [102.1.9]

'`lib/stdlib/environment.c`' [101.10.9]

'`lib/stdlib/setenv.c`' [101.10.17]

'`lib/stdlib/unsetenv.c`' [101.10.20]

VEDERE ANCHE

environ(7) [97.1], *getenv(3)* [94.51], *putenv(3)* [94.82].

94.95 os16: setpwent(3)

«

Vedere *getpwent(3)* [94.53].

94.96 os16: setvbuf(3)

Vedere *setbuf(3)* [94.93].

94.97 os16: snprintf(3)

Vedere *printf(3)* [94.78].

94.98 os16: sp(3)

Vedere *cs(3)* [94.12].

94.99 os16: sprintf(3)

Vedere *printf(3)* [94.78].

94.100 os16: srand(3)

Vedere *rand(3)* [94.85].

94.101 os16: ss(3)

Vedere *cs(3)* [94.12].

94.102 os16: sscanf(3)

Vedere *scanf(3)* [94.90].

94.103 os16: stdio(3)

NOME

‘**`stdio`**’ - libreria per la gestione dei file in forma di flussi di file (*stream*)

SINTASSI

```
#include <stdio.h>
```

DESCRIZIONE

Le funzioni di libreria che fanno capo al file di intestazione ‘`stdio.h`’, consentono di gestire i file in forma di «flussi», rappresentati da puntatori al tipo ‘**`FILE`**’. Questa gestione si sovrappone a quella dei file in forma di «descrittori», la quale avviene tramite chiamate di sistema. Lo scopo della sovrapposizione dovrebbe

essere quello di gestire i file con l'ausilio di una memoria tampone, cosa che però la libreria di os16 non fornisce.

Nella libreria di os16, il tipo '**`FILE *`**' è un puntatore a una variabile strutturata che contiene solo tre informazioni: il numero del descrittore del file a cui il flusso si associa; lo stato di errore; lo stato di raggiungimento della fine del file.

```
typedef struct {
    int             fdn;        // File descriptor number.
    char            error;      // Error indicator.
    char            eof;        // End of file indicator.
} FILE;
```

Le variabili strutturate necessarie per questa gestione, sono raccolte in un array, dichiarato nel file '`lib/stdio/FILE.c`', con il nome ***_stream[]***, dove per il descrittore di file ***n***, si associano sempre i dati di ***_stream[n]***.

```
FILE _stream[FOPEN_MAX];
```

Così come sono previsti tre descrittori (zero, uno e due) per la gestione di standard input, standard output e standard error, tutti i processi inizializzano l'array ***_stream[]*** con l'abbinamento a tali descrittori, per i primi tre flussi.

```
void
_stdio_stream_setup (void)
{
    _stream[0].fdn   = 0;
    _stream[0].error = 0;
    _stream[0].eof   = 0;

    _stream[1].fdn   = 1;
    _stream[1].error = 0;
    _stream[1].eof   = 0;

    _stream[2].fdn   = 2;
    _stream[2].error = 0;
    _stream[2].eof   = 0;
}
```

Ciò avviene attraverso il codice contenuto nel file '`crt0.s`', dove si chiama la funzione che provvede a tale inizializzazione, contenuta nel file '`lib/stdio/FILE.c`'. Per fare riferimento ai flussi predefiniti, si usano i nomi '**`stdin`**', '**`stdout`**' e '**`stderr`**', i quali sono dichiarati nel file '`stdio.h`', come puntatori ai primi tre elementi dell'array ***_stream[]***:

```
#define stdin   (&_stream[0])
#define stdout  (&_stream[1])
#define stderr  (&_stream[2])
```

FILE SORGENTI

'`lib/sys/types.h`' [101.14]

'`lib/stdio.h`' [101.9]

'`lib/stdio/FILE.c`' [101.9.1]

'`applic/crt0.s`' [102.1.9]

VEDERE ANCHE

close(2) [93.7], *open(2)* [93.28], *read(2)* [93.29], *write(2)* [93.44].

94.104 os16: strcat(3)

«

NOME

'**`strcat`**', '**`strncat`**' - concatenamento di una stringa a un'altra già esistente

SINTASSI

```
#include <string.h>
char *strcat  (char *restrict dst,
               const char *restrict org);
char *strncat (char *restrict dst,
               const char *restrict org,
               size_t n);
```

DESCRIZIONE

Le funzioni ***strcat()*** e ***strncat()*** copiano la stringa di origine ***org***, aggiungendola alla stringa di destinazione ***dst***, nel senso che la scrittura avviene a partire dal codice di terminazione '`\0`' che viene così sovrascritto. Al termine della copia, viene aggiunto nuovamente il codice di terminazione di stringa '`\0`', nella nuova posizione conclusiva.

Nel caso particolare di ***strncat()***, la copia si arresta al massimo dopo il trasferimento di ***n*** caratteri. Pertanto, la stringa di origine per ***strncat()*** potrebbe anche non essere terminata correttamente, se raggiunge o supera la dimensione di ***n*** caratteri. In ogni caso, nella destinazione viene aggiunto il codice nullo di terminazione di stringa, dopo la copia del carattere ***n***-esimo.

VALORE RESTITUITO

Le due funzioni restituiscono ***dst***.

FILE SORGENTI

'`lib/string.h`' [101.11]

'`lib/string/strcat.c`' [101.11.7]

'`lib/string/strncat.c`' [101.11.16]

VEDERE ANCHE

memccpy(3) [94.67], *memcpy(3)* [94.70], *strcpy(3)* [94.108], *strncpy(3)* [94.108].

94.105 os16: strchr(3)

«

NOME

'**`strchr`**', '**`strrchr`**' - ricerca di un carattere all'interno di una stringa

SINTASSI

```
#include <string.h>
char *strchr  (const char *string, int c);
char *strrchr (const char *string, int c);
```

DESCRIZIONE

Le funzioni ***strchr()*** e ***strrchr()*** scandiscono la stringa ***string*** alla ricerca di un carattere uguale al valore di ***c***. La funzione ***strchr()*** scandisce a partire da «sinistra», ovvero ricerca la prima corrispondenza con il carattere ***c***, mentre la funzione ***strrchr()*** cerca l'ultima corrispondenza con il carattere ***c***, pertanto è come se scandisse da «destra».

VALORE RESTITUITO

Se le due funzioni trovano il carattere che cercano, ne restituiscono il puntatore, altrimenti restituiscono '**`NULL`**'.

FILE SORGENTI

'`lib/string.h`' [101.11]

'`lib/string/strchr.c`' [101.11.8]

'`lib/string/strrchr.c`' [101.11.20]

VEDERE ANCHE

memchr(3) [94.68], *strlen(3)* [94.112], *strpbrk(3)* [94.116], *strspn(3)* [94.118].

94.106 os16: strcmp(3)

NOME

'**strcmp**', '**strncmp**' - confronto di due stringhe

SINTASSI

```
#include <string.h>
int strcmp  (const char *string1, const char *string2);
int strncmp (const char *string1, const char *string2,
              size_t n);
int strcoll (const char *string1, const char *string2);
```

DESCRIZIONE

Le funzioni ***strcmp()*** e ***strncmp()*** confrontano due stringhe, nel secondo caso, il confronto avviene al massimo fino al ***n***-esimo carattere.

La funzione ***strcoll()*** dovrebbe eseguire il confronto delle due stringhe tenendo in considerazione la configurazione locale. Tuttavia, os16 non è in grado di gestire le configurazioni locali, pertanto questa funzione coincide esattamente con ***strcmp()***.

VALORE RESTITUITO

Valore	Esito del confronto.
−1	***string1 < string2***
0	***string1 == string2***
+1	***string1 > string2***

FILE SORGENTI

'`lib/string.h`' [101.11]

'`lib/string/strcmp.c`' [101.11.9]

'`lib/string/strncmp.c`' [101.11.17]

'`lib/string/strcoll.c`' [101.11.10]

VEDERE ANCHE

memcmp(3) [94.69].

94.107 os16: strcoll(3)

«

Vedere *strcmp(3)* [94.106].

94.108 os16: strcpy(3)

«

NOME

‘**strcpy**’, ‘**strncpy**’ - copia di una stringa

SINTASSI

```
#include <string.h>
char *strcpy  (char *restrict dst,
               const char *restrict org);
char *strncpy (char *restrict dst,
               const char *restrict org,
               size_t n);
```

DESCRIZIONE

Le funzioni ***strcpy()*** e ***strncpy()***, copiano la stringa ***org***, completa di codice nullo di terminazione, nella destinazione ***dst***. Eventualmente, nel caso di ***strncpy()***, la copia non supera i primi ***n*** caratteri, con l'aggravante che in tal caso, se nei primi ***n*** caratteri non c'è il codice nullo di terminazione delle stringhe, nella destinazione ***dst*** si ottiene una stringa non terminata.

VALORE RESTITUITO

Le funzioni restituiscono ***dst***.

FILE SORGENTI

‘`lib/string.h`’ [101.11]

‘`lib/string/strcpy.c`’ [101.11.11]

‘`lib/string/strncpy.c`’ [101.11.18]

VEDERE ANCHE

memccpy(3) [94.67], *memcpy(3)* [94.70], *memmove(3)* [94.71].

94.109 os16: strcspn(3)

Vedere *strspn(3)* [94.118].

94.110 os16: strdup(3)

NOME

‘**strdup**’ - duplicazione di una stringa

SINTASSI

```
#include <string.h>
char *strdup  (const char *string);
```

DESCRIZIONE

La funzione ***strdup()***, alloca dinamicamente una quantità di memoria, necessaria a copiare la stringa ***string***, quindi esegue tale copia e restituisce il puntatore alla nuova stringa allocata. Tale puntatore può essere usato successivamente per liberare la memoria, con l’ausilio della funzione ***free()***.

VALORE RESTITUITO

La funzione restituisce il puntatore alla nuova stringa ottenuta dalla copia, oppure ‘**NULL**’ nel caso non fosse possibile allocare la memoria necessaria.

ERRORI

Valore di ***errno***	Significato
ENOMEM	Memoria insufficiente.

FILE SORGENTI

‘lib/string.h’ [101.11]

‘lib/string/strdup.c’ [101.11.13]

VEDERE ANCHE

free(3) [94.66], *malloc(3)* [94.66], *realloc(3)* [94.66].

94.111 os16: strerror(3)

«

NOME

'**strerror**' - descrizione di un errore in forma di stringa

SINTASSI

```
#include <string.h>
char *strerror (int errnum);
```

DESCRIZIONE

La funzione ***strerror()*** interpreta il valore ***errnum*** come un errore, di quelli che può rappresentare la variabile ***errno*** del file 'errno.h'.

VALORE RESTITUITO

La funzione restituisce il puntatore a una stringa contenente la descrizione dell'errore, oppure soltanto '**Unknown error**', se l'argomento ricevuto non è traducibile.

FILE SORGENTI

'lib/errno.h' [101.3]

'lib/string.h' [101.11]

'lib/string/strerror.c' [101.11.14]

VEDERE ANCHE

errno(3) [94.18], *perror(3)* [94.77].

94.112 os16: strlen(3)

«

NOME

'**strlen**' - lunghezza di una stringa

SINTASSI

```
#include <string.h>
size_t strlen (const char *string);
```

DESCRIZIONE

La funzione ***strlen()*** calcola la lunghezza della stringa, ovvero la quantità di caratteri che la compone, escludendo il codice nullo di conclusione.

VALORE RESTITUITO

La funzione restituisce la quantità di caratteri che compone la stringa, escludendo il codice '**\0**' finale.

FILE SORGENTI

'lib/string.h' [101.11]

'lib/string/strlen.c' [101.11.15]

94.113 os16: strncat(3)

Vedere *strcat(3)* [94.104].

94.114 os16: strncmp(3)

Vedere *strcmp(3)* [94.106].

94.115 os16: strncpy(3)

Vedere *strcpy(3)* [94.108].

94.116 os16: strpbrk(3)

NOME

'**strpbrk**' - scansione di una stringa alla ricerca di un carattere

SINTASSI

```
#include <string.h>
char *strpbrk (const char *string, const char *accept);
```

DESCRIZIONE

La funzione ***strpbrk()*** cerca il primo carattere, nella stringa ***string***, che corrisponda a uno di quelli contenuti nella stringa ***accept***.

VALORE RESTITUITO

Restituisce il puntatore al primo carattere che, nella stringa ***string*** corrisponde a uno di quelli contenuti nella stringa ***accept***. In mancanza di alcuna corrispondenza, restituisce '**NULL**'.

FILE SORGENTI

'lib/string.h' [101.11]

'lib/string/strpbrk.c' [101.11.19]

VEDERE ANCHE

memchr(3) [94.68], *strchr(3)* [94.105], *strstr(3)* [94.119], *strtok(3)* [94.120].

94.117 os16: strrchr(3)

Vedere *strchr(3)* [94.105].

94.118 os16: strspn(3)

NOME

'`strspn`', '`strcspn`' - scansione di una stringa, limitatamente a un certo insieme di caratteri

SINTASSI

```
#include <string.h>
size_t strspn  (const char *string, const char *accept);
size_t strcspn (const char *string, const char *reject);
```

DESCRIZIONE

La funzione ***strspn()*** scandisce la stringa ***string***, calcolando la lunghezza di questa che contiene, a partire dall'inizio, soltanto caratteri che si trovano nella stringa ***accept***.

La funzione ***strcspn()*** scandisce la stringa ***string***, calcolando la lunghezza di questa che contiene, a partire dall'inizio, soltanto caratteri che non si trovano nella stringa ***reject***.

VALORE RESTITUITO

La funzione ***strspn()*** restituisce la lunghezza della stringa che contiene soltanto caratteri contenuti in ***accept***.

La funzione ***strcspn()*** restituisce la lunghezza della stringa che contiene soltanto caratteri che non sono contenuti in ***reject***.

FILE SORGENTI

'`lib/string.h`' [101.11]

'`lib/string/strspn.c`' [101.11.21]

'`lib/string/strcspn.c`' [101.11.12]

VEDERE ANCHE

memchr(3) [94.68], *strchr(3)* [94.105], *strpbrk(3)* [94.116], *strstr(3)* [94.119], *strtok(3)* [94.120].

94.119 os16: strstr(3)

NOME

'**strstr**' - ricerca di una sottostringa

SINTASSI

```
#include <string.h>
char *strstr (const char *string, const char *substring);
```

DESCRIZIONE

La funzione ***strstr()*** scandisce la stringa ***string***, alla ricerca della prima corrispondenza con la stringa ***substring***, restituendo eventualmente il puntatore all'inizio di tale corrispondenza.

VALORE RESTITUITO

Se la ricerca termina con successo, viene restituito il puntatore all'inizio della sottostringa contenuta in ***string***; diversamente viene restituito il puntatore nullo '**NULL**'.

FILE SORGENTI

'lib/string.h' [101.11]

'lib/string/strstr.c' [101.11.22]

VEDERE ANCHE

memchr(3) [94.68], *strchr(3)* [94.105], *strpbrk(3)* [94.116], *strtok(3)* [94.120].

94.120 os16: strtok(3)

NOME

'**strtok**' - *string token*, ovvero estrazione di pezzi da una stringa

SINTASSI

```
#include <string.h>
char *strtok (char *restrict string,
              const char *restrict delim);
```

DESCRIZIONE

La funzione ***strtok()*** serve a suddividere una stringa in unità, definite *token*, specificando un elenco di caratteri da intendere come delimitatori, in una seconda stringa. La funzione va usata in fasi successive, fornendo solo inizialmente la stringa da suddividere che continua poi a essere utilizzata se al suo posto viene fornito il puntatore nullo. La funzione restituisce, di volta in volta, il puntatore alla sottostringa contenente l'unità individuata, oppure il puntatore nullo, se non può trovarla.

La funzione deve tenere memoria di un puntatore in modo persistente e deve isolare le unità modificando la stringa originale, inserendo il carattere nullo di terminazione alla fine delle unità individuate.

Quando la funzione viene chiamata indicando al posto della stringa da scandire il puntatore nullo, l'insieme dei delimitatori può essere diverso da quello usato nelle fasi precedenti.

Per comprendere lo scopo della funzione viene utilizzato lo stesso esempio che appare nel documento *ISO/IEC 9899:TC2*, al paragrafo 7.21.5.7, con qualche piccola modifica per poterlo rendere un programma autonomo:

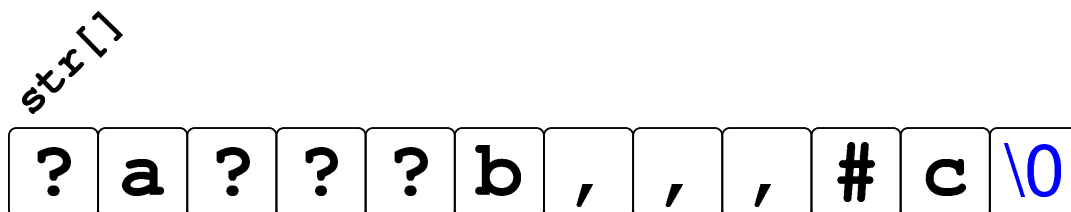
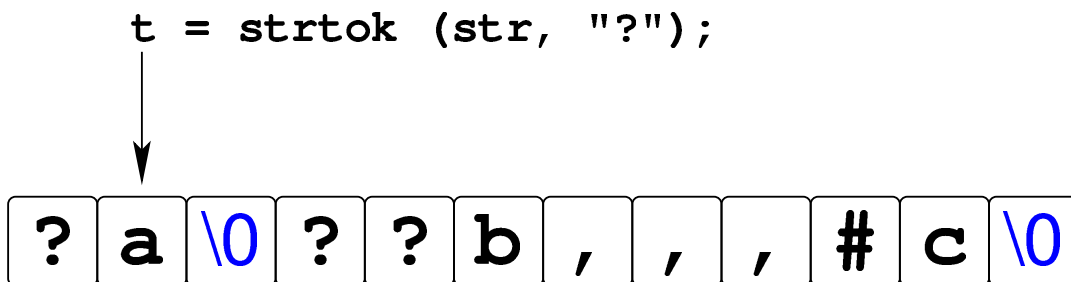
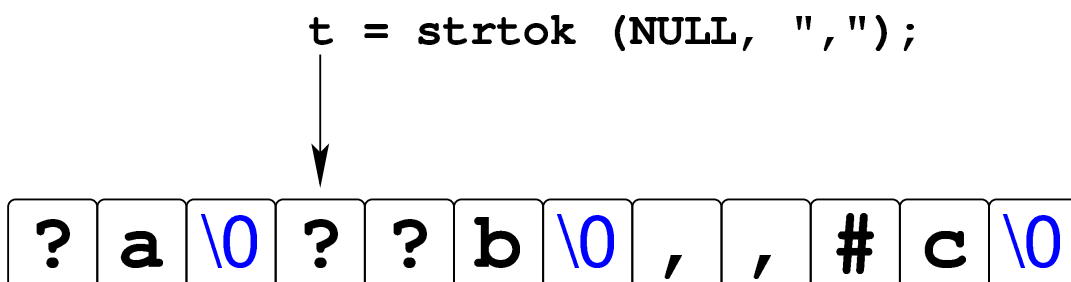
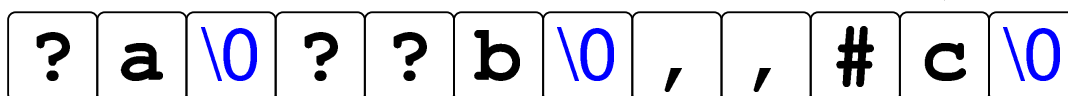
```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char str[] = "?a???b,,,#c";
  char *t;

  t = strtok (str, "?");         // t punta all'unità "a"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, ",");        // t punta all'unità "??b"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, "#,");       // t punta all'unità "c"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, "?");        // t è un puntatore nullo
  printf ("strtok: \"%s\"\n", t);

  return 0;
}
```

Avviando il programma si ottiene quanto già descritto dai commenti inseriti nel codice:

```
strtok: "a"
strtok: "??b"
strtok: "c"
strtok: "(null)"
```

Ciò che avviene nell'esempio può essere schematizzato come segue. Inizialmente la stringa '**`str`**' ha in memoria l'aspetto seguente:

Dopo la prima chiamata della funzione ***strtok()*** la stringa risulta alterata e il puntatore ottenuto raggiunge la lettera '**`a`**':

Dopo la seconda chiamata della funzione, in cui si usa il puntatore nullo per richiedere una scansione ulteriore della stringa originale, si ottiene un nuovo puntatore che, questa volta, inizia a partire dal quarto carattere, rispetto alla stringa originale, dal momento che il terzo è già stato sovrascritto da un carattere nullo:

La penultima chiamata della funzione ***strtok()*** raggiunge la lettera '**`c`**' che è anche alla fine della stringa originale:

L'ultimo tentativo di chiamata della funzione non può dare alcun esito, perché la stringa originale si è già conclusa.

VALORE RESTITUITO

La funzione restituisce il puntatore al prossimo «pezzo«, oppure '**NULL**' se non ce ne sono più.

FILE SORGENTI

'lib/string.h' [101.11]

'lib/string/strtok.c' [101.11.23]

VEDERE ANCHE

memchr(3) [94.68], *strchr(3)* [94.105], *strpbrk(3)* [94.116], *strspn(3)* [94.118].

94.121 os16: strtol(3)

«

NOME

'**strtol**', '**strtoul**' - conversione di una stringa in un numero

SINTASSI

```
#include <stdlib.h>
long int strtol (const char *restrict string,
                 char **restrict endptr,
                 int base);


unsigned long int strtoul (const char *restrict string,
                           char **restrict endptr,
                           int base);
```

DESCRIZIONE

Le funzioni ***strtol()*** e ***strtoul()***, convertono la stringa ***string*** in un numero, intendendo la sequenza di caratteri nella base di numerazione indicata come ultimo argomento (***base***). Tuttavia, la base di numerazione potrebbe essere omessa (valore zero) e in tal caso la stringa deve essere interpretata ugualmente in qualche modo: se (dopo un segno eventuale) inizia con zero seguito da un'altra cifra numerica, deve trattarsi di una sequenza ottale; se inizia con zero, quindi appare una lettera «x» deve trattarsi di un numero esadecimale; se inizia con una cifra numerica diversa da zero, deve trattarsi di un numero in base dieci.

La traduzione della stringa ha luogo progressivamente, arrestandosi quando si incontra un carattere incompatibile con la base di numerazione selezionata o stabilita automaticamente. Il valore convertito viene restituito; inoltre, se il puntatore ***endptr*** è valido (diverso da '**`NULL`**'), si assegna a ****endptr*** la posizione raggiunta nella stringa, corrispondente al primo carattere che non può essere convertito. Pertanto, nello stesso modo, se la stringa non può essere convertita affatto e si può assegnare qualcosa a ****endptr***, alla fine, ****endptr*** corrisponde esattamente a ***string***.

VALORE RESTITUITO

Le funzioni restituiscono il valore tratto dall'interpretazione della stringa, ammesso che sia rappresentabile, altrimenti si ottiene '**`LONG_MIN`**' o '**`LONG_MAX`**', a seconda dei casi, sapendo che occorre consultare la variabile ***errno*** per maggiori dettagli.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Argomento non valido.
`ERANGE`	Il valore risultante è al di fuori dell'intervallo ammissibile per la rappresentazione.

DIFETTI

La realizzazione di ***strtoul()*** è incompleta, in quanto si limita a utilizzare ***strtol()***, convertendo il risultato in un valore senza segno.

FILE SORGENTI

'`lib/stdlib.h`' [101.10]

'`lib/stdlib/strtol.c`' [101.10.18]

'`lib/stdlib/strtoul.c`' [101.10.19]

94.122 os16: strtoul(3)

Vedere *strtol(3)* [94.121].

94.123 os16: strxfrm(3)

NOME

'**strxfrm**' - *string transform*, ovvero trasformazione di una stringa

SINTASSI

```
#include <string.h>
size_t strxfrm (char *restrict dst,
                const char *restrict org,
                size_t n);
```

DESCRIZIONE

Lo scopo della funzione ***strxfrm()*** sarebbe quello di copiare la stringa ***org***, sovrascrivendo ***dst***, fino a un massimo di ***n*** caratteri nella destinazione, ma applicando una trasformazione relativa alla configurazione locale.

os16 non gestisce la configurazione locale, pertanto questa funzione si comporta in modo simile a ***strncpy()***, con una differenza in ciò che viene restituito.

VALORE RESTITUITO

La funzione restituisce la quantità di byte utilizzati per contenere la trasformazione in ***dst***, senza però contare il carattere nullo di terminazione.

FILE SORGENTI

'lib/string.h' [101.11]

'lib/string/strxfrm.c' [101.11.24]

VEDERE ANCHE

memcmp(3) [94.69], *strcmp(3)* [94.106], *strcoll(3)* [94.106].

94.124 os16: ttyname(3)

NOME

'**ttyname**' - determinazione del percorso del file di dispositivo di un terminale aperto

SINTASSI

```
#include <unistd.h>
char *ttyname (int fdn);
```

DESCRIZIONE

La funzione ***ttyname()*** richiede come unico argomento il numero che identifica il descrittore di un file. Ammesso che tale descrittore si riferisca a un terminale, la funzione restituisce il puntatore a una stringa che rappresenta il percorso del file di dispositivo corrispondente.

La stringa in questione viene modificata se si usa la funzione in altre occasioni.

VALORE RESTITUITO

La funzione restituisce il puntatore a una stringa che descrive il percorso del file di dispositivo, presunto, del terminale aperto con il numero ***fdn***. Se non si tratta di un terminale, si ottiene un errore. In ogni caso, se la funzione non può restituire un'informazione corretta, produce semplicemente il puntatore nullo e aggiorna la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file indicato non è valido.
`ENOTTY`	Il descrittore di file indicato non riguarda un terminale.

FILE SORGENTI

'`lib/unistd.h`' [101.17]

'`lib/unistd/ttyname.c`' [101.17.34]

VEDERE ANCHE

stat(2) [93.36], *isatty(3)* [94.61].

94.125 os16: unsetenv(3)

Vedere *setenv(3)* [94.94].

94.126 os16: vfprintf(3)

Vedere *vprintf(3)* [94.128].

94.127 os16: vfscanf(3)

Vedere *vfscanf(3)* [94.129].

94.128 os16: vprintf(3)

NOME

'**vprintf**', '**vfprintf**', '**vsprintf**', '**vsnprintf**' - composizione dei dati per la visualizzazione

SINTASSI

```
#include <stdarg.h>
#include <stdio.h>
int vprintf   (char *restrict format, va_list arg);
int vfprintf  (FILE *fp, char *restrict format,
               va_list arg);
int vsnprintf (char *restrict string, size_t size,
               const char *restrict format, va_list ap);
int vsprintf  (char *string, char *restrict format,
               va_list arg);
```

DESCRIZIONE

Le funzioni del gruppo '**v...printf()**' hanno in comune lo scopo di comporre dei dati in forma di stringa, generalmente per la visualizzazione, o comunque per la fruizione a livello umano.

I dati in ingresso possono essere vari e vengono comunicati attraverso un puntatore di tipo '**va_list**'. Per quantificare e qualificare questi dati in ingresso, la stringa a cui punta il parametro ***format***, deve contenere degli ***specificatori di conversione***, oltre eventualmente ad altri caratteri. Pertanto, queste funzioni, prendono la stringa a cui punta ***format***, la interpretano e determinano come scandire gli argomenti a cui fa riferimento il puntatore ***arg***, quindi producono un'altra stringa, composta dalla stringa precedente, sostituendo gli specificatori di conversione con i dati a cui questi si riferiscono, secondo una forma di conversione definita dagli specificatori stessi.

In generale, le funzioni '**v...printf()**' servono per realizzare le altre funzioni '**...printf()**', le quali invece ricevono gli argomenti variabili direttamente. Per esempio, la funzione ***printf()*** può essere realizzata utilizzando in pratica ***vprintf()***:

```
#include <stdio.h>
#include <stdarg.h>
int
printf (char *restrict format, ...)
{
    va_list ap;
    va_start (ap, format);
    return (vprintf (format, ap));
}
```

Si veda *printf(3)* [94.78], per la descrizione di come va predisposta la stringa ***format***. Nella realizzazione di os16, di tutte queste funzioni, quella che compie effettivamente il lavoro di interpretazione della stringa di formato e che in qualche modo viene chiamata da tutte le altre, è soltanto ***vsnprintf()***.

VALORE RESTITUITO

Le funzioni restituiscono la quantità di caratteri utilizzati nella composizione della nuova stringa, escluso il carattere nullo di terminazione.

FILE SORGENTI

'`lib/stdarg.h`' [101.1.12]

'`lib/stdio.h`' [101.9]

'`lib/stdio/FILE.c`' [101.9.1]

'`lib/stdio/vfprintf.c`' [101.9.36]

'`lib/stdio/vprintf.c`' [101.9.39]

'`lib/stdio/vsprintf.c`' [101.9.42]

'`lib/stdio/vsnprintf.c`' [101.9.41]

VEDERE ANCHE

fprintf(3) [94.78], *printf(3)* [94.78], *sprintf(3)* [94.78], *snprintf(3)* [94.78], *scanf(3)* [94.90].

94.129 os16: vscanf(3)

«

NOME

'**vscanf**', '**vfscanf**', '**vsscanf**' - interpretazione dell'input e conversione

SINTASSI

```
#include <stdarg.h>
#include <stdio.h>
int vscanf  (const char *restrict format, va_list ap);
int vfscanf (FILE *restrict fp, const char *restrict format,
             va_list ap);
int vsscanf (const char *string, const char *restrict format,
             va_list ap);
```

DESCRIZIONE

Le funzioni del gruppo '**v...scanf()**' hanno in comune lo scopo di interpretare dei dati, forniti in forma di stringa, convertendoli opportunamente.

I dati in ingresso sono costituiti da una sequenza di caratteri, la quale viene fornita tramite lo standard input per ***vscanf()***, tramite il flusso di file ***fp*** per ***vfscanf()***, oppure tramite la stringa ***string*** per ***vsscanf()***. L'interpretazione dei dati in ingresso viene guidata da una stringa di formato, costituita dal parametro ***format***, per le tre funzioni. La stringa di formato contiene degli ***specificatori di conversione***, con cui si determina il tipo degli argomenti variabili a cui punta inizialmente ***ap***.

Queste funzioni servono per realizzare in pratica quelle corrispondenti che hanno nomi privi della lettera «v» iniziale. Per esempio, per ottenere ***scanf()*** si può utilizzare ***vscanf()***:

```
#include <stdio.h>
#include <stdarg.h>
int
scanf (const char *restrict format, ...)
{
    va_list ap;
    va_start (ap, format);
    return vscanf (format, ap);
}
```

Il modo in cui va predisposta la stringa di formato (***format***) è descritto in *scanf(3)* [94.90]. La funzione più importante di questo gruppo, in quanto svolge effettiva-

mente il lavoro di interpretazione e viene chiamata, più o meno indirettamente, da tutte le altre, è ***vfsscanf()***, la quale però non è standard.

VALORE RESTITUITO

Le funzioni restituiscono la quantità di elementi in ingresso interpretati e assegnati correttamente: una quantità inferiore al previsto indica pertanto un errore. Se le funzioni restituiscono il valore '**`EOF`**', si tratta di un errore, dovuto eventualmente a un problema di interpretazione del formato o a un problema di accesso al flusso di file da cui deve provenire l'input.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
`EPERM`	Operazione di accesso non consentita.
`EACCES`	Accesso non consentito.
`EBADF`	Il descrittore del file a cui si riferisce il flusso, non è valido.
`ERANGE`	Il risultato della conversione di un intero, non può essere memorizzato nel tipo di variabile a cui si riferisce lo specificatore di conversione.

FILE SORGENTI

'`lib/stdio.h`' [101.9]

'`lib/stdio/vfscanf.c`' [101.9.37]

'`lib/stdio/vscanf.c`' [101.9.40]

'`lib/stdio/vsscanf.c`' [101.9.43]

'`lib/stdio/vfsscanf.c`' [101.9.38]

VEDERE ANCHE

fscanf(3) [94.90], *scanf(3)* [94.90], *sscanf(3)* [94.90], *printf(3)* [94.78].

94.130 os16: vsnprintf(3)

«

Vedere *vprintf(3)* [94.128].

94.131 os16: vsprintf(3)

«

Vedere *vprintf(3)* [94.128].

94.132 os16: vsscanf(3)

«

Vedere *vsscanf(3)* [94.129].

Sezione 4: file speciali

TXT
htm
PDF
pdf
P1
P2
P3
P4
epub

95.1 os16: console(4)

NOME

'/dev/console' - file di dispositivo che rappresenta la console e le console virtuali

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'/dev/console'	file di dispositivo a caratteri	5	255	0644_8
'/dev/console0'	file di dispositivo a caratteri	5	0	0644_8
'/dev/console1'	file di dispositivo a caratteri	5	1	0644_8
'/dev/console2'	file di dispositivo a caratteri	5	2	0644_8
'/dev/console3'	file di dispositivo a caratteri	5	3	0644_8

DESCRIZIONE

Il file di dispositivo '/dev/console' rappresenta la console virtuale attiva in un certo momento; i file '/etc/console***n***' rappresentano la console virtuale ***n***, dove ***n*** va da zero a quattro.

I permessi di accesso a questi file di dispositivo sono limitati in modo da consentire solo al proprietario di accedere in scrittura. Tuttavia, per i file di dispositivo usati effettivamente come terminali di controllo, i permessi e la proprietà sono gestiti automaticamente dai programmi '**getty**' e '**login**'.

VEDERE ANCHE

MAKEDEV(8) [98.3], *tty(4)* [95.11].

95.2 os16: dsk(4)

NOME

'/dev/dsk***n***' - file di dispositivo per le unità di memorizzazione a disco

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'/dev/dsk0'	file di dispositivo a blocchi	3	0	0644_8
'/dev/dsk1'	file di dispositivo a blocchi	3	1	0644_8
'/dev/dsk2'	file di dispositivo a blocchi	3	2	0644_8
'/dev/dsk3'	file di dispositivo a blocchi	3	3	0644_8

DESCRIZIONE

I file di dispositivo '/dev/dsk***n***' rappresentano, ognuno, un'unità di memorizzazione a disco. La prima unità è '/dev/dsk0', quelle successive procedono con la numerazione.

os16 gestisce solo unità a dischetti da 1440 Kibyte; inoltre, non è ammissibile la suddivisione in partizioni e, in pratica, sono gestibili solo due unità. Pertanto, sono utili solo '/dev/dsk0' e '/dev/dsk1'.

VEDERE ANCHE

MAKEDEV(8) [98.3].

95.3 os16: kmem_file(4)

«

NOME

'/dev/kmem_file' - accesso alla memoria del kernel contenente la tabella dei file

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'/dev/kmem_file'	file di dispositivo a caratteri	4	5	0444_8

DESCRIZIONE

Il file di dispositivo '/dev/kmem_file' consente di accedere in lettura all'area di memoria che, nel kernel, rappresenta la tabella dei file. La tabella dei file è un array di '**FILE_MAX_SLOTS**' elementi, di tipo '**file_t**', secondo le definizioni contenute nel file 'kernel/fs.h'.

VEDERE ANCHE

MAKEDEV(8) [98.3], *kmem_ps(4)* [95.6], *kmem_mmp(4)* [95.5], *kmem_sb(4)* [95.7], *kmem_inode(4)* [95.4].

95.4 os16: kmem_inode(4)

NOME

'`/dev/kmem_inode`' - accesso alla memoria del kernel contenente la tabella degli inode

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'`/dev/kmem_inode`'	file di dispositivo a caratteri	4	4	0444_8

DESCRIZIONE

Il file di dispositivo '`/dev/kmem_inode`' consente di accedere in lettura all'area di memoria che, nel kernel, rappresenta la tabella degli inode. La tabella degli inode è un array di '**`INODE_MAX_SLOTS`**' elementi, di tipo '**`inode_t`**', secondo le definizioni contenute nel file '`kernel/fs.h`'.

VEDERE ANCHE

MAKEDEV(8) [98.3], *kmem_ps(4)* [95.6], *kmem_mmp(4)* [95.5], *kmem_sb(4)* [95.7], *kmem_file(4)* [95.3].

95.5 os16: kmem_mmp(4)

NOME

'`/dev/kmem_mmp`' - accesso alla memoria del kernel contenente la mappa di utilizzo della memoria

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'`/dev/kmem_mmp`'	file di dispositivo a caratteri	4	2	0444_8

DESCRIZIONE

Il file di dispositivo '`/dev/kmem_mmp`' consente di accedere in lettura all'area di memoria che, nel kernel, rappresenta la mappa di utilizzo della memoria.

VEDERE ANCHE

MAKEDEV(8) [98.3], *kmem_ps(4)* [95.6], *kmem_sb(4)* [95.7], *kmem_inode(4)* [95.4], *kmem_file(4)* [95.3].

95.6 os16: kmem_ps(4)

«

NOME

'/dev/kmem_ps' - accesso alla memoria del kernel contenente lo stato dei processi

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'/dev/kmem_ps'	file di dispositivo a caratteri	4	1	0444_8

DESCRIZIONE

Il file di dispositivo '/dev/kmem_ps' consente di accedere in lettura all'area di memoria che, nel kernel, rappresenta la tabella dei processi. La tabella dei processi è un array di '**PROCESS_MAX**' elementi, di tipo '**proc_t**', secondo le definizioni contenute nel file 'kernel/proc.h'. Questo meccanismo viene usato dal programma '**ps**' per leggere e visualizzare lo stato dei processi.

VEDERE ANCHE

MAKEDEV(8) [98.3], *kmem_mmp(4)* [95.5], *kmem_sb(4)* [95.7], *kmem_inode(4)* [95.4], *kmem_file(4)* [95.3].

95.7 os16: kmem_sb(4)

«

NOME

'/dev/kmem_sb' - accesso alla memoria del kernel contenente la tabella dei super blocchi

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'/dev/kmem_sb'	file di dispositivo a caratteri	4	3	0444_8

DESCRIZIONE

Il file di dispositivo '`/dev/kmem_sb`' consente di accedere in lettura all'area di memoria che, nel kernel, rappresenta la tabella dei super blocchi. La tabella dei super blocchi è un array di '**`SB_MAX_SLOTS`**' elementi, di tipo '**`sb_t`**', secondo le definizioni contenute nel file '`kernel/fs.h`'.

VEDERE ANCHE

MAKEDEV(8) [98.3], *kmem_ps(4)* [95.6], *kmem_mmp(4)* [95.5], *kmem_inode(4)* [95.4], *kmem_file(4)* [95.3].

95.8 os16: mem(4)

«

NOME

'`/dev/mem`' - file di dispositivo per l'acceso alla memoria del processo

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'`/dev/mem`'	file di dispositivo a caratteri	1	1	0444_8

DESCRIZIONE

Il file di dispositivo '`/dev/mem`' consente di leggere la memoria del processo.

VEDERE ANCHE

MAKEDEV(8) [98.3].

95.9 os16: null(4)

«

NOME

'`/dev/null`' - file di dispositivo per la distruzione dei dati

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'`/dev/null`'	file di dispositivo a caratteri	1	2	0666_8

DESCRIZIONE

Il file di dispositivo '`/dev/null`' appare in lettura come un file completamente vuoto, mentre in scrittura è un file in cui si può scrivere indefinitivamente, perdendo però i dati che vi si immettono.

VEDERE ANCHE

MAKEDEV(8) [98.3], *zero(4)* [95.12].

95.10 os16: port(4)

NOME

'`/dev/port`' - file di dispositivo per accedere alle porte di I/O

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'`/dev/port`'	file di dispositivo a caratteri	1	3	0644_8

DESCRIZIONE

Il file di dispositivo '`/dev/port`' consente di accedere alle porte di I/O. Tali porte consentono di leggere uno o al massimo due byte, pertanto la dimensione della lettura può essere '**`(size_t) 1`**' oppure '**`(size_t) 2`**'. Per selezionare l'indirizzo della porta occorre posizionare il riferimento interno al file a un indirizzo pari a quello della porta, prima di eseguire la lettura o la scrittura.

VEDERE ANCHE

MAKEDEV(8) [98.3], *mem(4)* [95.8].

95.11 os16: tty(4)

NOME

'`/dev/tty`' - file di dispositivo che rappresenta il terminale di controllo del processo

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'`/dev/tty`'	file di dispositivo a caratteri	2	0	0666_8

DESCRIZIONE

Il file di dispositivo '`/dev/tty`' rappresenta il terminale di controllo del processo; in altri termini, il processo che accede al file '`/dev/tty`', raggiunge il proprio terminale di controllo.

VEDERE ANCHE

MAKEDEV(8) [98.3], *console(4)* [95.1].

95.12 os16: zero(4)

«

NOME

'`/dev/zero`' - file di dispositivo per la produzione del valore zero

CONFIGURAZIONE

File	Tipo	Numero primario	Numero secondario	Permessi
'`/dev/zero`'	file di dispositivo a caratteri	1	4	0666_8

DESCRIZIONE

Il file di dispositivo '`/dev/zero`' appare in lettura come un file di lunghezza indefinita, contenente esclusivamente il valore zero (lo zero binario), mentre in scrittura è un file in cui si può scrivere indefinitivamente, perdendo però i dati che vi si immettono.

VEDERE ANCHE

MAKEDEV(8) [98.3], *null(4)* [95.9].

Sezione 5: formato dei file e convenzioni

TXT htm PDF pdf P1 P2 P3 P4 epub

96.1 os16: inittab(5)

NOME

'/etc/inittab' - configurazione di '**init**'

DESCRIZIONE

Il file '/etc/inittab' contiene la configurazione di '**init**', per la definizione dei processi da avviare per la messa in funzione del sistema operativo. Il file può contenere dei commenti, preceduti dal carattere «#» e «voci» costituite da righe suddivise in quattro campi, separati da due punti (:), come nell'esempio seguente:

```
c0:1:respawn:/bin/getty /dev/console0
c1:1:respawn:/bin/getty /dev/console1
```

I campi hanno il significato descritto nell'elenco seguente:

1. codice che identifica univocamente la voce;
2. i livelli di esecuzione per cui la voce è valida;
3. l'azione da compiere sulla voce;
4. il programma da avviare, con tutte le opzioni e gli argomenti necessari.

Il programma '**init**' di os16 non distingue i livelli di esecuzione e considera soltanto l'azione '**respawn**', con la quale si intende che '**init**' debba riavviare il processo, quando questo muore, o comunque quando muore quel processo che ha preso il suo posto.

VEDERE ANCHE

init(8) [98.2], *getty(8)* [98.1], *login(1)* [92.12].

96.2 os16: issue(5)

NOME

'/etc/issue' - messaggio che precede '**login**'

DESCRIZIONE

Il file '/etc/issue' viene visualizzato da '**getty**', prima dell'avvio di '**login**'. Il contenuto predefinito di questo file, per os16, è il seguente:

```
os16: a basic os. [Ctrl q], [Ctrl r], [Ctrl s], [Ctrl t] to change console.
```

Il programma '**getty**' di os16 non è in grado di interpretare il contenuto del file, pertanto lo visualizza letteralmente; tuttavia, '**getty**' mostra, indipendentemente dalla presenza e dal contenuto del file '`/etc/issue`', delle informazioni sul terminale per il quale è in funzione.

VEDERE ANCHE

getty(8) [98.1].

96.3 os16: passwd(5)

«

NOME

'`/etc/passwd`' - elenco delle utenze

DESCRIZIONE

Il file '`/etc/passwd`' contiene l'elenco degli utenti del sistema, uno per ogni riga. Le righe sono divise in sette campi, delimitati con il carattere due punti (:), come nell'esempio seguente, che rappresenta l'impostazione predefinita di os16:

```
root:ciao:0:0:root:/root:/bin/shell
user:ciao:1001:1001:test user:/home/user:/bin/shell
```

I campi hanno il significato descritto nell'elenco seguente:

1. nominativo utente;
2. parola d'ordine, in chiaro, per l'identificazione con il programma '**login**';
3. numero UID, ovvero il numero dell'utente;
4. numero GID, ovvero il numero del gruppo, ma non utilizzato da os16;
5. descrizione dell'utenza;
6. shell.

Trattandosi di un sistema operativo elementare, la parola d'ordine appare in chiaro nel secondo campo, senza altri accorgimenti. Inoltre, il file deve essere accessibile in lettura a tutti gli utenti.

VEDERE ANCHE

login(1) [92.12].

Sezione 7: varie

Capitolo 97

97.1 os16: environ(7)

NOME

'**environ**' - ambiente del processo elaborativo

SINTASSI

```
extern char **environ;
```

DESCRIZIONE

La variabile ***environ***, dichiarata nel file 'unistd.h', punta a un array di stringhe, ognuna delle quali rappresenta una variabile di ambiente, con il valore a lei assegnato. Pertanto, il contenuto di queste stringhe ha una forma del tipo '***nome*=*valore***'. Per esempio '**HOME=/home/user**'.

In generale, l'accesso diretto ai contenuti di questo array non è conveniente, in quanto sono disponibili delle funzioni che facilitano la gestione di questi dati in forma di variabili di ambiente.

Dal momento che le funzioni di accesso alle informazioni sulle variabili di ambiente sono definite nel file 'stdlib.h', la gestione effettiva dell'array di stringhe a cui punta ***environ*** è inserita nei file contenuti nella directory 'lib/stdlib/' di os16. Per la precisione, nel file 'lib/stdlib/environment.c' si dichiara l'array di caratteri ***_environment_table[][]*** e array di puntatori a caratteri ***_environment[]***:

```
char  _environment_table[ARG_MAX/32][ARG_MAX/16];
char *_environment[ARG_MAX/32+1];
```

L'array ***_environment_table[][]*** viene inizializzato con lo stato delle variabili di ambiente ereditate con l'avvio del processo; inoltre, all'array ***_environment[]*** vengono assegnati i puntatori alle varie stringhe che si possono estrapolare da ***_environment_table[]***. Questo lavoro iniziale avviene per opera della funzione ***_environment_setup()***, la quale viene avviata a sua volta dal file 'crt0.s'. Successivamente, nello stesso file 'crt0.s', viene copiato l'indirizzo dell'***_environment[]*** nella variabile ***environ***, di cui sopra.

FILE SORGENTI

'lib/unistd.h' [101.17]

'lib/stdlib.h' [101.10]

'lib/unistd/environ.c' [101.17.8]

‘`applic/crt0.s`’ [102.1.9]

‘`lib/stdlib/environment.c`’ [101.10.9]

‘`lib/stdlib/getenv.c`’ [101.10.11]

‘`lib/stdlib/putenv.c`’ [101.10.14]

‘`lib/stdlib/setenv.c`’ [101.10.17]

‘`lib/stdlib/unsetenv.c`’ [101.10.20]

VEDERE ANCHE

getenv(3) [94.51], *putenv(3)* [94.82], *setenv(3)* [94.94], *unsetenv(3)* [94.94].

97.2 os16: undocumented(7)

Questa sezione ha il solo scopo di raccogliere i riferimenti ipertestuali dei listati che, per qualche ragione, sono privi di una documentazione specifica.

Capitolo 98

Sezione 8: comandi per l'amministrazione del sistema

TXT htm PDF pdf P1 P2 P3 P4 epub

98.1 os16: getty(8)

NOME

'**`getty`**' - predisposizione di un terminale e avvio di '**`login`**'

SINTASSI

```
getty terminale
```

DESCRIZIONE

Il programma '**`getty`**' viene avviato da '**`init`**' per predisporre il terminale, ripristinando anche i permessi predefiniti, e per avviare successivamente il programma '**`login`**'. Prima di avviare '**`login`**', '**`getty`**' visualizza il contenuto del file '`/etc/issue`', se disponibile, inoltre mostra almeno l'indicazione del terminale attuale. Va osservato che questa realizzazione di '**`getty`**' lascia a '**`login`**' il compito di chiedere l'inserimento del nominativo utente.

FILE

'`/etc/issue`'

'**`getty`**' visualizza il contenuto di questo file prima di avviare '**`login`**'.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/getty.c`' [102.1.12]

VEDERE ANCHE

login(1) [92.12], *issue(5)* [96.2].

98.2 os16: init(8)

NOME

'**`init`**' - progenitore di tutti gli altri processi

SINTASSI

```
init
```

DESCRIZIONE

Il programma '**init**' viene avviato dal kernel (deve trattarsi precisamente del file '/bin/init') come primo e unico processo figlio del kernel stesso. Pertanto, '**init**' deve assumere il numero PID uno.

Questa realizzazione di '**init**' si limita a leggere il file '/etc/inittab' per determinare quali programmi figli avviare, senza poter distinguere da diversi livelli di esecuzione. In pratica, all'interno di questo file si indica l'uso di '**getty**', per la gestione dei terminali disponibili.

FILE

'/etc/inittab'
: Contiene l'indicazione dei processi che '**init**' deve avviare.

DIFETTI

Con os16 non è possibile associare ai segnali un'azione diversa da quella predefinita; quindi '**init**' non può essere informato dell'intenzione di arrestare il sistema. Pertanto, tale funzionalità non è stata realizzata nella versione di '**init**' di os16.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'applic/init.c' [102.1.13]

VEDERE ANCHE

inittab(5) [96.1].

98.3 os16: MAKEDEV(8)

«

NOME

'**MAKEDEV**' - creazione dei file di dispositivo

SINTASSI

```
MAKEDEV
```

DESCRIZIONE

'**MAKEDEV**' è un programma che crea, nella directory corrente, tutti i file di dispositivo previsti per os16. Tali file devono trovarsi normalmente nella directory '/dev/', pertanto, prima di usare '**MAKEDEV**' è necessario che la directory corrente corrisponda precisamente a tale posizione.

OPZIONI

Non sono previste opzioni per l'uso di '**MAKEDEV**', dal momento che vengono creati tutti i file di dispositivo, considerato il loro numero estremamente limitato.

NOTE

Tradizionalmente '**MAKEDEV**' viene realizzato in forma di script, ma os16 non dispone di una shell adeguata e non è possibile utilizzare script.

FILE SORGENTI

'applic/crt0.s' [102.1.9]

'lib/sys/os16.h' [101.12]

'applic/MAKEDEV.c' [102.1.1]

98.4 os16: mount(8)

«

NOME

'**mount**', '**umount**' - innesto e distacco di un file system

SINTASSI

mount *dispositivo* *dir_innesto* [*opzioni*]

umount *directory*

DESCRIZIONE

'**mount**' innesta il file system contenuto nell'unità di memorizzazione rappresentata dal file di dispositivo che va indicato come primo argomento, nella directory che appare come secondo argomento. Eventualmente si possono specificare delle opzioni di innesto, come terzo argomento.

'**umount**' stacca il file system innestato precedentemente nella directory indicata come unico argomento del comando.

OPZIONI DI INNESTO

Opzione	Descrizione
ro	Innesta il file system in sola lettura.
rw	Innesta il file system in lettura e scrittura. Si tratta comunque del comportamento predefinito, in mancanza di un'opzione contraria.

DIFETTI

Non viene preso in considerazione un eventuale file '`/etc/fstab`'; inoltre, l'utente non può conoscere lo stato degli innesti già in essere e, a questo proposito, l'uso di '**`mount`**' senza argomenti produce semplicemente un errore.

FILE SORGENTI

'`applic/crt0.s`' [102.1.9]

'`applic/mount.c`' [102.1.21]

'`applic/umount.c`' [102.1.27]

98.5 os16: umount(8)

Vedere *mount(8)* [98.4].

Sezione 9: kernel

TXT
htm
PDF
pdf
P1
P2
P3
P4
epub

99.1 os16: devices(9)

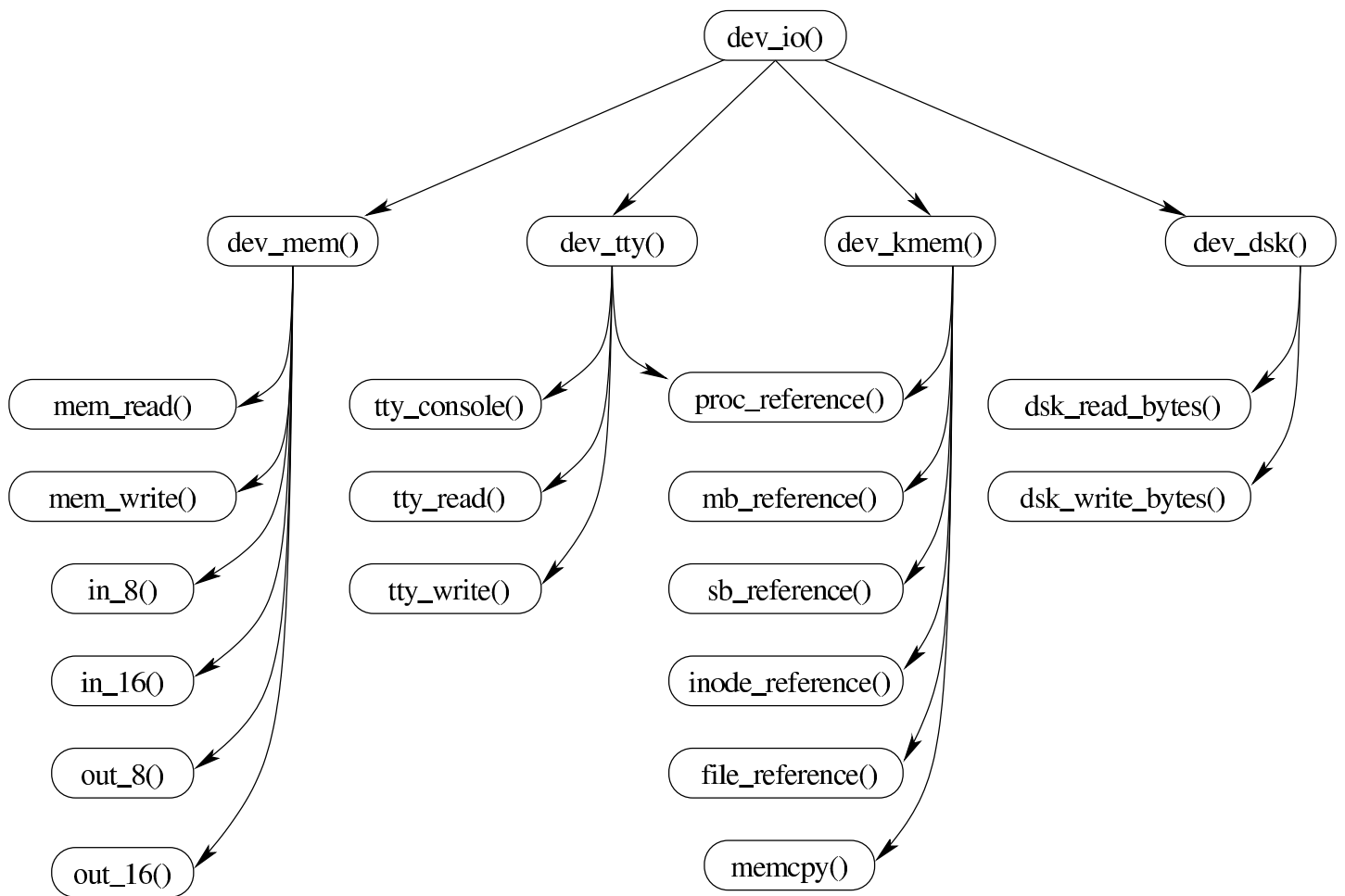
Il file '`kernel/devices.h`' [100.2] descrive ciò che serve per la gestione dei dispositivi. Tuttavia, la definizione dei numeri di dispositivo è contenuta nel file '`lib/sys/os16.h`' [101.12], il quale viene incluso da '`devices.h`'.

Tabella 90.41. Classificazione dei dispositivi di os16.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_MEM`	r/w	diretto	Permette l'accesso alla memoria, in modo indiscriminato, perché os16 non offre alcun tipo di protezione al riguardo.
`DEV_NULL`	r/w	nessuno	Consente la lettura e la scrittura, ma non si legge e non si scrive alcunché.
`DEV_PORT`	r/w	sequenziale	Consente di leggere e scrivere da o verso una porta di I/O, individuata attraverso l'indirizzo di accesso (l'indirizzo, o meglio lo scostamento, viene trattato come la porta a cui si vuole accedere). Tuttavia, la dimensione dell'informazione da trasferire è valida solo se si tratta di uno o di due byte: per la dimensione di un byte si usano le funzioni ***in_8()*** e ***out_8()***; per due byte si usano le funzioni ***in_16()*** e ***out_16()***. Per dimensioni differenti la lettura o la scrittura non ha effetto.
`DEV_ZERO`	r	sequenziale	Consente solo la lettura di valori a zero (zero inteso in senso binario).
`DEV_TTY`	r/w	sequenziale	Rappresenta il terminale virtuale del processo attivo.
`DEV_DSK`***n***	r/w	diretto	Rappresenta l'unità a dischi ***n***. os16 non gestisce le partizioni.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_KMEM_PS`	r	diretto	Rappresenta la tabella contenente le informazioni sui processi. L'indirizzo di accesso indica il numero del processo di partenza; la dimensione da leggere dovrebbe essere abbastanza grande da contenere un processo, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_KMEM_MMP`	r	sequenziale	Rappresenta la mappa della memoria, alla quale si può accedere solo dal suo principio. In pratica, l'indirizzo di accesso viene ignorato, mentre conta solo la quantità di byte richiesta.
`DEV_KMEM_SB`	r	diretto	Rappresenta la tabella dei super blocchi (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il super blocco; la dimensione richiesta dovrebbe essere abbastanza grande da contenere un super blocco, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_KMEM_INODE`	r	diretto	Rappresenta la tabella degli inode (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare l'inode; la dimensione richiesta dovrebbe essere abbastanza grande da contenere un inode, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_KMEM_FILE`	r	diretto	Rappresenta la tabella dei file (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il file; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di un file, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_CONSOLE`	r/w	sequenziale	Legge o scrive relativamente alla console attiva la quantità di byte richiesta, ignorando l'indirizzo di accesso.
`DEV_CONSOLE`***n***	r/w	sequenziale	Legge o scrive relativamente alla console ***n*** la quantità di byte richiesta, ignorando l'indirizzo di accesso.

Figura 90.38. Interdipendenza tra la funzione ***dev_io()*** e le altre. I collegamenti con le funzioni ***major()*** e ***minor()*** sono omesse.

99.1.1 os16: dev_io(9)

«

NOME

'**`dev_io`**' - interfaccia di accesso ai dispositivi

SINTASSI

```
<kernel/devices.h>
ssize_t dev_io (pid_t pid, dev_t device, int rw, off_t offset,
                void *buffer, size_t size, int *eof);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`dev_t` ***device***	Dispositivo, in forma numerica.
`int` ***rw***	Può assumere i valori '**`DEV_READ`**' o '**`DEV_WRITE`**', per richiedere rispettivamente un accesso in lettura oppure in scrittura.
`off_t` ***offset***	Posizione per l'accesso al dispositivo.
`void` *****buffer***	Memoria tampone, per la lettura o la scrittura.
`size_t` ***size***	Quantità di byte da leggere o da scrivere.
`int` *****eof***	Puntatore a una variabile in cui annotare, eventualmente, il raggiungimento della fine del file.

DESCRIZIONE

La funzione ***dev_io()*** è un'interfaccia generale per l'accesso ai dispositivi gestiti da os16.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letti o scritti effettivamente. In caso di errore restituisce il valore -1 e aggiorna la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`ENODEV`	Il numero del dispositivo non è valido.
`EIO`	Errore di input-output.

FILE SORGENTI

'`kernel/devices.h`' [100.2]

'`kernel/devices/dev_io.c`' [100.2.2]

VEDERE ANCHE

dev_dsk(9) [99.1.2], *dev_kmem(9)* [99.1.3], *dev_mem(9)* [99.1.4], *dev_tty(9)* [99.1.5].

99.1.2 os16: dev_dsk(9)

«

NOME

'**dev_dsk**' - interfaccia di accesso alle unità di memorizzazione di massa

SINTASSI

```
<kernel/devices.h>
ssize_t dev_dsk (pid_t pid, dev_t device, int rw,
                 off_t offset, void *buffer,
                 size_t size, int *eof);
```

DESCRIZIONE

La funzione *dev_dsk()* consente di accedere alle unità di memorizzazione di massa, che per os16 si riducono ai soli dischetti da 1440 Kibyte.

Per il significato degli argomenti, il valore restituito e gli eventuali errori, si veda *dev_io(9)* [99.1.1].

FILE SORGENTI

'kernel/devices.h' [100.2]

'kernel/devices/dev_io.c' [100.2.2]

'kernel/devices/dev_dsk.c' [100.2.1]

99.1.3 os16: dev_kmem(9)

«

NOME

'**dev_kmem**' - interfaccia di accesso alle tabelle di dati del kernel, rappresentate in memoria

SINTASSI

```
<kernel/devices.h>
ssize_t dev_kmem (pid_t pid, dev_t device, int rw,
                  off_t offset,
                  void *buffer, size_t size, int *eof);
```

DESCRIZIONE

La funzione ***dev_kmem()*** consente di accedere, solo in lettura, alle porzioni di memoria che il kernel utilizza per rappresentare alcune tabelle importanti. Per poter interpretare ciò che si ottiene occorre riprodurre la struttura di un elemento della tabella a cui si è interessati, pertanto occorre incorporare il file di intestazione del kernel che la descrive.

Dispositivo	Tabella a cui si riferisce
`DEV_KMEM_PS`	Si accede alla tabella dei processi, all'elemento rappresentato da ***offset***: ***proc_table[offset]***.
`DEV_KMEM_MMP`	Si accede alla mappa della memoria, ovvero la tabella ***mb_table[]***, senza considerare il valore di ***offset***.
`DEV_KMEM_SB`	Si accede alla tabella dei super blocchi, all'elemento rappresentato da ***offset***: ***sb_table[offset]***.
`DEV_KMEM_INODE`	Si accede alla tabella degli inode, all'elemento rappresentato da ***offset***: ***inode_table[offset]***.
`DEV_KMEM_FILE`	Si accede alla tabella dei file di sistema, all'elemento rappresentato da ***offset***: ***file_table[offset]***.

Per il significato degli argomenti della chiamata, per interpretare il valore restituito e gli eventuali errori, si veda *dev_io(9)* [99.1.1].

FILE SORGENTI

'`kernel/devices.h`' [100.2]

'`kernel/devices/dev_io.c`' [100.2.2]

'`kernel/devices/dev_kmem.c`' [100.2.3]

99.1.4 os16: dev_mem(9)

NOME

'**`dev_mem`**' - interfaccia di accesso alla memoria, in modo indiscriminato

SINTASSI

```
<kernel/devices.h>
ssize_t dev_mem (pid_t pid, dev_t device, int rw,
                 off_t offset, void *buffer,
                 size_t size, int *eof);
```

DESCRIZIONE

La funzione ***dev_mem()*** consente di accedere, in lettura e in scrittura alla memoria e alle porte di input-output.

Dispositivo	Descrizione
`DEV_MEM`	Si tratta della memoria centrale, complessiva, dove il valore di ***offset*** rappresenta l'indirizzo efficace (complessivo) a partire da zero.
`DEV_NULL`	Si tratta di ciò che realizza il file di dispositivo tradizionale '`/dev/null`': la scrittura si perde semplicemente e la lettura non dà alcunché.
`DEV_ZERO`	Si tratta di ciò che realizza il file di dispositivo tradizionale '`/dev/zero`': la scrittura si perde semplicemente e la lettura produce byte a zero (zero binario).
`DEV_PORT`	Consente l'accesso alle porte di input-output. La dimensione rappresentata da ***size*** può essere solo pari a uno o due: una dimensione pari a uno richiede di comunicare un solo byte con una certa porta; una dimensione pari a due richiede la comunicazione di un valore a 16 bit. Il valore di ***offset*** serve a individuare la porta di input-output con cui si intende comunicare (leggere o scrivere un valore).

Per quanto non viene descritto qui, si veda *dev_io(9)* [99.1.1].

FILE SORGENTI

'`kernel/devices.h`' [100.2]

'`kernel/devices/dev_io.c`' [100.2.2]

'`kernel/devices/dev_mem.c`' [100.2.4]

99.1.5 os16: dev_tty(9)

«

NOME

'**`dev_tty`**' - interfaccia di accesso alla console

SINTASSI

```
<kernel/devices.h>
ssize_t dev_tty (pid_t pid, dev_t device, int rw, off_t offset,
                 void *buffer, size_t size, int *eof);
```

DESCRIZIONE

La funzione ***dev_tty()*** consente di accedere, in lettura e in scrittura, a una console virtuale, scelta in base al numero del dispositivo.

Quando la lettura richiede l'attesa per l'inserimento da tastiera, se il processo elaborativo ***pid*** non è il kernel, allora viene messo in pausa, in attesa di un evento legato al terminale.

Il sistema di gestione del terminale è molto povero con os16. Va osservato che il testo letto viene anche visualizzato automaticamente. Quando un processo non vuole mostrare il testo sullo schermo, deve provvedere a sovrascriverlo immediatamente, facendo arretrare il cursore preventivamente.

Per quanto non viene descritto qui, si veda *dev_io(9)* [99.1.1].

FILE SORGENTI

'`kernel/devices.h`' [100.2]

'`kernel/devices/dev_io.c`' [100.2.2]

'`kernel/devices/dev_tty.c`' [100.2.5]

99.2 os16: diag(9)

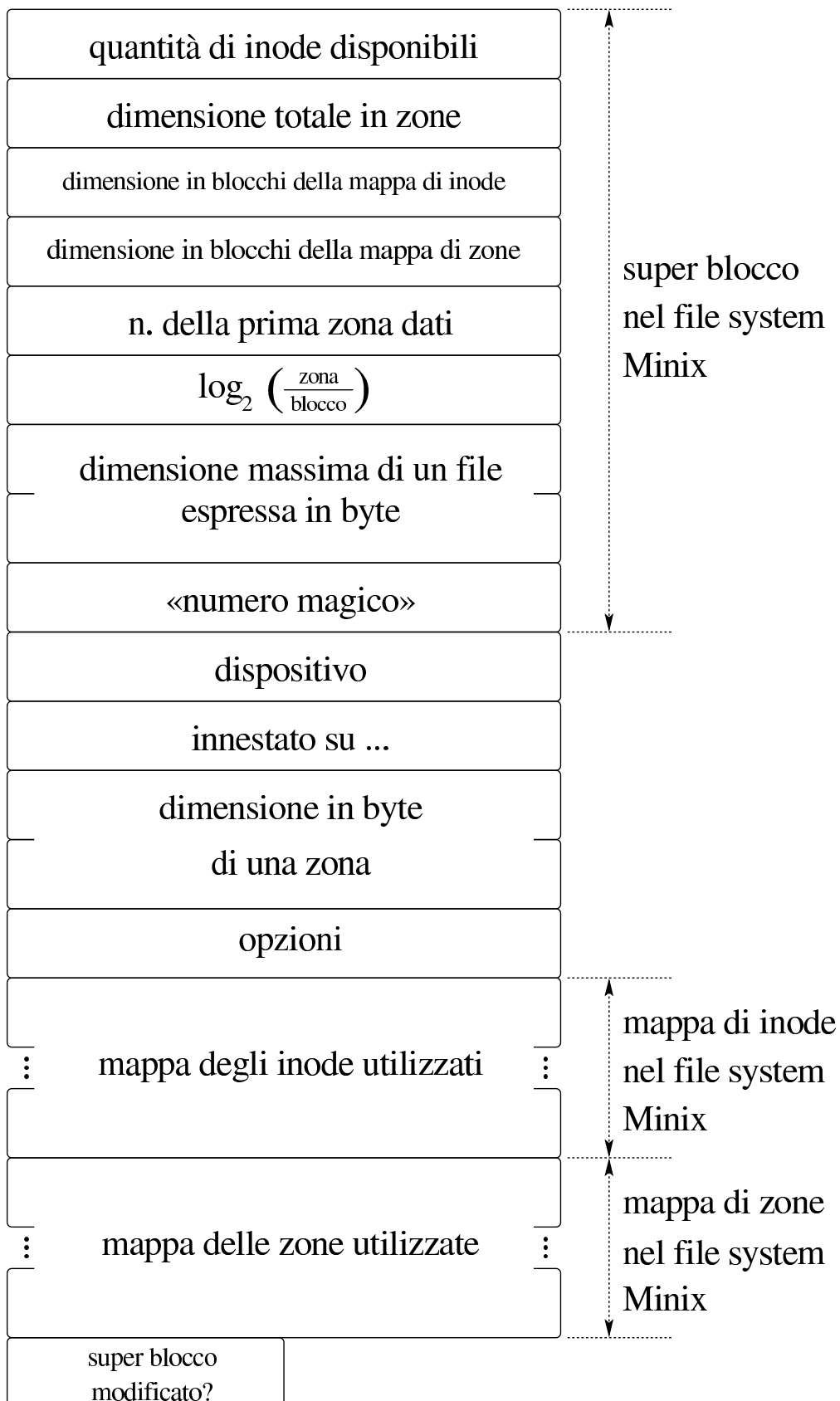
Il file '`kernel/diag.h`' [100.3] descrive alcune funzioni e macro-istruzioni, per uso diagnostico. Lo scopo di queste è di mostrare o di rendere visualizzabile alcune informazioni interne alla gestione del kernel.

Alcune di queste funzioni sono usate, altre no. Per esempio durante il funzionamento interattivo del kernel vengono usate ***print_proc_list()***, ***print_segments()***, ***print_kmem()***, ***print_time()*** e ***print_mb_map()***.

99.3 os16: fs(9)

Il file '`kernel/fs.h`' [100.4] descrive ciò che serve per la gestione del file system, che per os16 corrisponde al tipo Minix 1.

La gestione del file system, a livello complessivo di sistema, è suddivisa in tre aspetti principali: super blocco, inode e file. Per ognuno di questi è prevista una tabella (di super blocchi, di inode e di file). Seguono delle figure che descrivono l'organizzazione di queste tabelle.

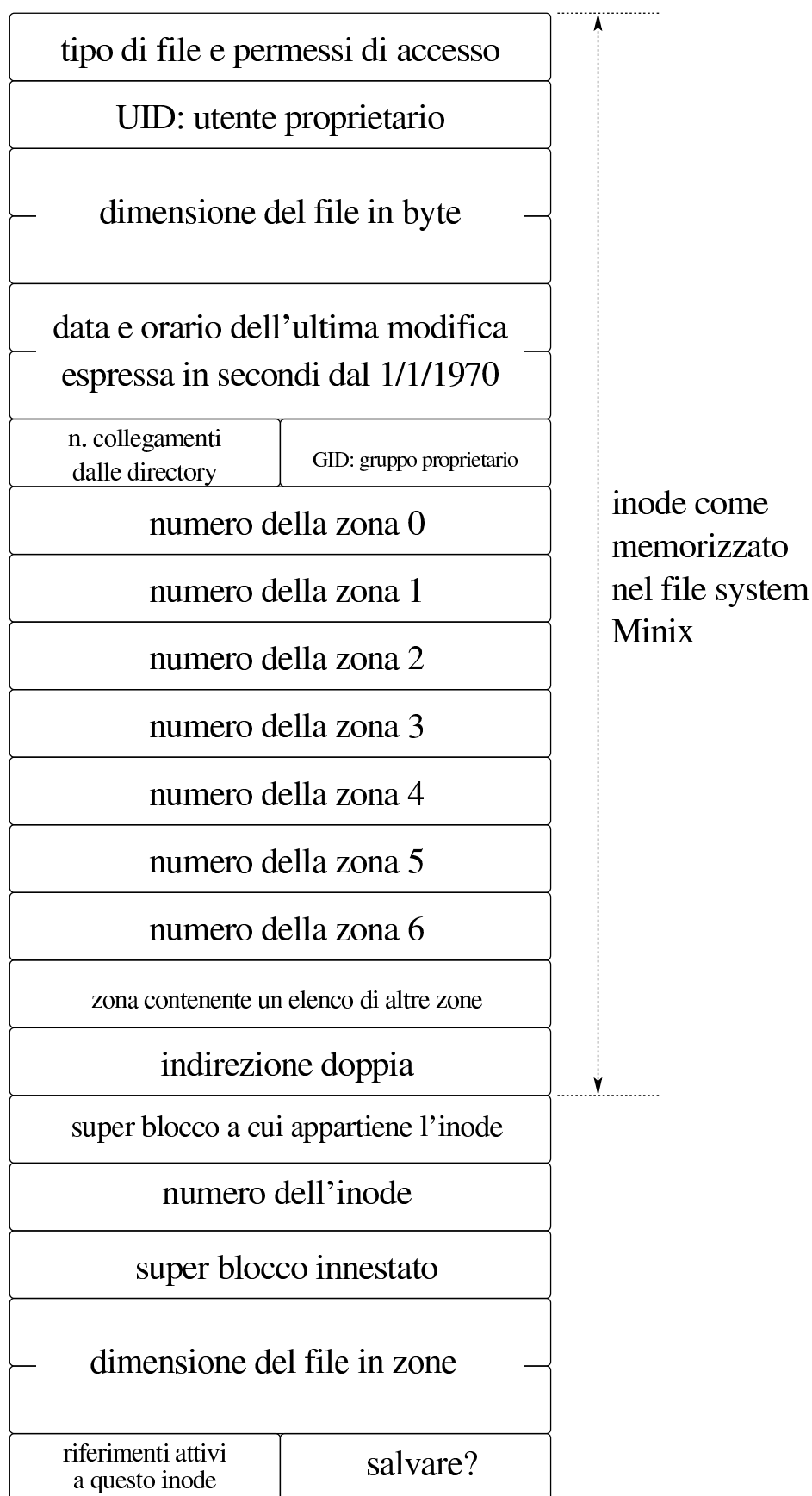
Figura 90.42. Struttura del tipo '`sb_t`', corrispondente agli elementi dell'array ***sb_table[]***.

Listato 90.43. Struttura del tipo '**`sb_t`**', corrispondente agli elementi dell'array ***sb_table[]***.

```
typedef struct sb        sb_t;

struct sb {
  uint16_t  inodes;
  uint16_t  zones;
  uint16_t  map_inode_blocks;
  uint16_t  map_zone_blocks;
  uint16_t  first_data_zone;
  uint16_t  log2_size_zone;
  uint32_t  max_file_size;
  uint16_t  magic_number;
  //-------------------------
  dev_t     device;
  inode_t  *inode_mounted_on;
  blksize_t blksize;
  int       options;
  uint16_t  map_inode[SB_MAP_INODE_SIZE];
  uint16_t  map_zone[SB_MAP_ZONE_SIZE];
  char      changed;
};
```

Figura 90.47. Struttura del tipo '**inode_t**', corrispondente agli elementi dell'array ***inode_table[]***.

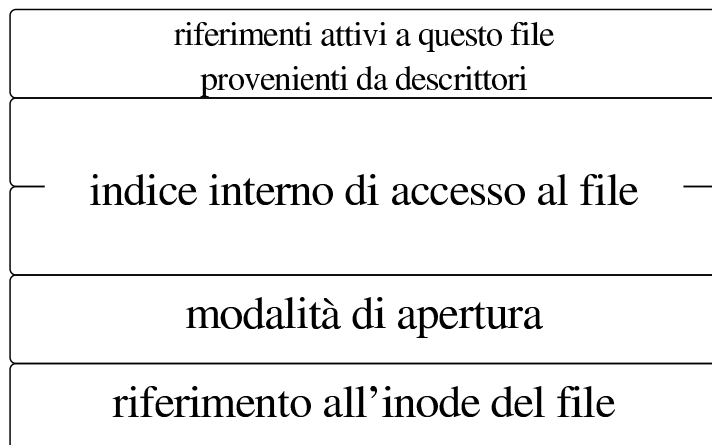
Campo
tipo di file e permessi di accesso
UID: utente proprietario
dimensione del file in byte
data e orario dell'ultima modifica espressa in secondi dal 1/1/1970
n. collegamenti dalle directory / GID: gruppo proprietario
numero della zona 0
numero della zona 1
numero della zona 2
numero della zona 3
numero della zona 4
numero della zona 5
numero della zona 6
zona contenente un elenco di altre zone
indirezione doppia
super blocco a cui appartiene l'inode
numero dell'inode
super blocco innestato
dimensione del file in zone
riferimenti attivi a questo inode / salvare?

inode come memorizzato nel file system Minix

Listato 90.48. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.

```
<verbatimpre width="60">
<![CDATA[
typedef struct inode     inode_t;

struct inode {
    mode_t         mode;
    uid_t          uid;
    ssize_t        size;
    time_t         time;
    uint8_t        gid;
    uint8_t        links;
    zno_t          direct[7];
    zno_t          indirect1;
    zno_t          indirect2;
    //------------------------
    sb_t          *sb;
    ino_t          ino;
    sb_t          *sb_attached;
    blkcnt_t       blkcnt;
    unsigned char references;
    char           changed;
};
```

Figura 90.54. Struttura del tipo '**`file_t`**', corrispondente agli elementi dell'array ***file_table[]***.

riferimenti attivi a questo file provenienti da descrittori
indice interno di accesso al file
modalità di apertura
riferimento all'inode del file

```
typedef struct file file_t;

struct file {
  int       references;
  off_t     offset;
  int       oflags;
  inode_t *inode;
};
```

Figura 90.57. Struttura del tipo '**`fd_t`**', con cui si costituiscono gli elementi delle tabelle dei descrittori di file, una per ogni processo.

indicatori dello stato del file e delle modalità di accesso
indicatori del descrittore
riferimento alla tabella dei file di sistema

```
typedef struct fd fd_t;
struct fd {
    int      fl_flags;
    int      fd_flags;
    file_t *file;
};
```

99.3.1 os16: fd_chmod(9)

«

NOME

'**`fd_chmod`**' - cambiamento della modalità dei permessi di un descrittore di file

SINTASSI

```
<kernel/fs.h>
int fd_chmod (pid_t pid, int fdn, mode_t mode);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`int` ***fdn***	Numero del descrittore di file.
`mode_t` ***mode***	Modalità dei permessi, di cui si prendono in considerazione solo i 12 bit meno significativi.

DESCRIZIONE

La funzione ***fs_chmod()*** cambia la modalità dei permessi del file aperto con il descrittore numero ***fdn***, secondo il valore contenuto nel parametro ***mode***, di cui però si considerano solo gli ultimi 12 bit. L'operazione viene svolta per conto del processo ***pid***, il quale deve avere i privilegi necessari per poter intervenire così. La modifica della modalità dei permessi raggiunge l'inode del file a cui fa capo il descrittore in questione; pertanto l'inode viene necessariamente salvato dopo la modifica. Il fatto che il descrittore di file possa essere stato aperto in sola lettura, non impedisce la modifica dell'inode attuata da questa funzione.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**SYS_FCHMOD**'. Nella libreria standard, si avvale di questa funzionalità *fchmod(2)* [93.4].

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Operazione fallita, con aggiornamento della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file ***fdn*** non è valido.
`EACCES`	Il processo elaborativo non ha i privilegi necessari nei confronti del file.

FILE SORGENTI

'`lib/sys/stat/fchmod.c`' [101.13.2]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/fd_chmod.c`' [100.4.1]

VEDERE ANCHE

fchmod(2) [93.4], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7].

99.3.2 os16: fd_chown(9)

NOME

'**fd_chown**' - cambiamento della proprietà di un descrittore di file

SINTASSI

```
<kernel/fs.h>
int fd_chown (pid_t pid, int fdn, uid_t uid, gid_t gid);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`int` ***fdn***	Numero del descrittore di file.
`uid_t` ***uid***	Nuova proprietà che si intende attribuire al file.
`gid_t` ***gid***	Nuovo gruppo che si intenderebbe attribuire al file.

DESCRIZIONE

La funzione ***fs_chown()*** cambia la proprietà del file già aperto, individuato attraverso il suo descrittore. L'operazione viene svolta per conto del processo ***pid***, il quale deve avere i privilegi necessari per poter intervenire così: in pratica deve trattarsi di un processo con identità efficace pari a zero, perché os16 non considera la gestione dei gruppi. La modifica della proprietà raggiunge l'inode del file a cui fa capo il descrittore in questione; pertanto l'inode viene necessariamente salvato dopo la modifica. Il fatto che il descrittore di file possa essere stato aperto in sola lettura, non impedisce la modifica dell'inode attuata da questa funzione.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_FCHOWN`**'. Nella libreria standard, si avvale di questa funzionalità *fchown(2)* [93.4].

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Operazione fallita, con aggiornamento della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file ***fdn*** non è valido.
`EACCES`	Il processo elaborativo non ha i privilegi necessari nei confronti del file.

FILE SORGENTI

'`lib/unistd/fchown.c`' [101.17.16]

'`kernel/proc.h`' [100.9]

'kernel/proc/_isr.s' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

'kernel/fs.h' [100.4]

'kernel/fs/fd_chown.c' [100.4.2]

VEDERE ANCHE

fchown(2) [93.5], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7].

99.3.3 os16: fd_close(9)

NOME

'**fd_close**' - chiusura di un descrittore di file

SINTASSI

```
<kernel/fs.h>
int fd_close (pid_t pid, int fdn);
```

ARGOMENTI

Argomento	Descrizione
pid_t ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
int ***fdn***	Numero del descrittore di file.

DESCRIZIONE

La funzione ***fd_close()*** chiude il descrittore di file specificato come argomento. Per ottenere questo risultato, oltre che intervenire nella tabella dei descrittori associata al processo elaborativo specificato come argomento, riduce il contatore dei riferimenti nella voce corrispondente della tabella dei file; se però questo contatore raggiunge lo zero, anche l'inode viene liberato, attraverso *inode_put(9)* [99.3.24].

Questa funzione viene usata in modo particolare da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**SYS_CLOSE**'. Nella libreria standard, si avvale di questa funzionalità *close(2)* [93.7]. La funzione '**fd_close**' è comunque usata internamente al kernel, in tutte le occasioni in cui la chiusura di un descrittore deve avvenire in modo implicito.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Operazione fallita, con aggiornamento della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file ***fdn*** non è valido.

FILE SORGENTI

'`lib/unistd/close.c`' [101.17.5]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/fd_close.c`' [100.4.3]

VEDERE ANCHE

close(2) [93.7], *sysroutine(9)* [99.8.28], *inode_put(9)* [99.3.24].

99.3.4 os16: fd_dup(9)

«

NOME

'**`fd_dup`**', '**`fd_dup2`**' - duplicazione di un descrittore di file

SINTASSI

```
<kernel/fs.h>
int fd_dup  (pid_t pid, int fdn_old, int fdn_min);
int fd_dup2 (pid_t pid, int fdn_old, int fdn_new);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`int` ***fdn_old***	Numero del descrittore di file da duplicare.
`int` ***fdn_min***	Primo numero di descrittore da usare per la copia.
`int` ***fdn_new***	Numero di descrittore da usare per la copia.

DESCRIZIONE

Le funzioni ***fd_dup()*** e ***fd_dup2()*** duplicano un descrittore, nel senso che sdoppiano l'accesso a un file in due descrittori. La funzione ***fd_dup()***, per il duplicato da realizzare, cerca un descrittore libero, cominciando da ***fdn_min*** e continuando progressivamente, fino al primo disponibile. La funzione ***fd_dup2()***, invece, richiede di specificare esattamente il descrittore da usare per il duplicato, con la differenza che, se ***fdn_new*** è già utilizzato, prima della duplicazione viene chiuso.

In entrambi i casi, il descrittore ottenuto dalla copia, viene privato dell'indicatore '**`FD_CLOEXEC`**', ammesso che nel descrittore originale ci fosse.

Queste funzioni vengono usate da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_DUP`**' e '**`SYS_DUP2`**'. Inoltre, la funzione *fd_fcntl(9)* [99.3.6] si avvale di ***fd_dup()*** per la duplicazione di un descrittore. Le funzioni della libreria standard che si avvalgono delle chiamate di sistema che poi raggiungono ***fd_dup()*** e ***fd_dup2()*** sono *dup(2)* [93.8] e *dup2(2)* [93.8].

VALORE RESTITUITO

Le due funzioni restituiscono il numero del descrittore prodotto dalla duplicazione. In caso di errore, invece, restituiscono il valore −1, aggiornando la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il valore di ***fdn_min*** o ***fdn_new*** è impossibile.
`EBADF`	Il valore di ***fdn_old*** non è valido.
`EMFILE`	Non è possibile allocare un nuovo descrittore.

FILE SORGENTI

'lib/unistd/dup.c' [101.17.6]

'lib/unistd/dup2.c' [101.17.7]

'kernel/proc.h' [100.9]

'kernel/proc/_isr.s' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

'kernel/fs.h' [100.4]

'kernel/fs/fd_dup.c' [100.4.4]

'kernel/fs/fd_dup2.c' [100.4.5]

VEDERE ANCHE

dup(2) [93.8], *dup2(2)* [93.8], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7].

99.3.5 os16: fd_dup2(9)

Vedere *fd_dup(9)* [99.3.4].

99.3.6 os16: fd_fcntl(9)

NOME

'**fd_fcntl**' - configurazione e intervento sui descrittori di file

SINTASSI

```
<kernel/fs.h>
int fd_fcntl (pid_t pid, int fdn, int cmd, int arg);
```

ARGOMENTI

Argomento	Descrizione
pid_t ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
int ***fdn***	Numero del descrittore a cui si fa riferimento.
int ***cmd***	Comando di ***fd_fcntl()***.
int ***arg***	Argomento eventuale del comando.

DESCRIZIONE

La funzione ***fd_fcntl()*** esegue un'operazione, definita dal parametro ***cmd***, sul descrittore ***fdn***. A seconda del tipo di operazione richiesta, può essere preso in considerazione anche l'argomento corrispondente al parametro ***arg***. Il valore del parametro ***cmd*** che rappresenta l'operazione richiesta, va fornito in forma di costante simbolica, come descritto nell'elenco seguente. Tali macro-variabili derivano dalle dichiarazioni contenute nel file '`lib/sys/fcntl.h`'.

Sintassi	Descrizione
`fd_fcntl (`***pid***`, `***fdn***`, F_DUPFD, (int) `***fdn_min***`)`	Richiede la duplicazione del descrittore di file ***fdn***, in modo tale che la copia abbia il numero di descrittore minore possibile, ma maggiore o uguale a quello indicato come argomento ***fdn_min***.
`fd_fcntl (`***pid***`, `***fdn***`, F_GETFD, 0)` `fd_fcntl (`***pid***`, `***fdn***`, F_SETFD, (int) `***fd_flags***`)`	Rispettivamente, legge o imposta, gli indicatori del descrittore di file ***fdn*** (eventualmente noti come *fd flags*). È possibile impostare un solo indicatore, '**`FD_CLOEXEC`**', pertanto, al posto di ***fd_flags*** si può mettere solo la costante '**`FD_CLOEXEC`**'.
`fd_fcntl (`***pid***`, `***fdn***`, F_GETFL, 0)` `fd_fcntl (`***pid***`, `***fdn***`, F_SETFL, (int) `***fl_flags***`)`	Rispettivamente, legge o imposta, gli indicatori dello stato del file, relativi al descrittore ***fdn***. Per impostare questi indicatori, vanno combinate delle costanti simboliche: '**`O_RDONLY`**', '**`O_WRONLY`**', '**`O_RDWR`**', '**`O_CREAT`**', '**`O_EXCL`**', '**`O_NOCTTY`**', '**`O_TRUNC`**'.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_FCNTL`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***fd_fcntl()*** è *fcntl(2)* [93.13].

VALORE RESTITUITO

Il significato del valore restituito dalla funzione dipende dal tipo di operazione richiesta, come sintetizzato dalla tabella successiva.

Operazione richiesta	Significato del valore restituito
`F_DUPFD`	Si ottiene il numero del descrittore prodotto dalla copia, oppure −1 in caso di errore.
`F_GETFD`	Si ottiene il valore degli indicatori del descrittore (*fd flags*, oppure −1 in caso di errore.
`F_GETFL`	Si ottiene il valore degli indicatori del file (*fl flags*, oppure −1 in caso di errore.

`F_GETOWN`
`F_SETOWN`
`F_GETLK`
`F_SETLK`
`F_SETLKW`
	Si ottiene −1, in quanto si tratta di operazioni non realizzate in questa versione della funzione, per os16.

altri tipi di operazione	Si ottiene 0 in caso di successo, oppure −1 in caso di errore.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il descrittore di file non è valido.
`EINVAL`	È stato richiesto un tipo di operazione non valido.
`EMFILE`	Non è possibile duplicare il descrittore, perché non ce ne sono di liberi.

FILE SORGENTI

‘`lib/fcntl/fcntl.c`’ [101.4.2]

‘`kernel/proc.h`’ [100.9]

‘`kernel/proc/_isr.s`’ [100.9.1]

‘`kernel/proc/sysroutine.c`’ [100.9.30]

‘`kernel/fs.h`’ [100.4]

‘`kernel/fs/fd_fcntl.c`’ [100.4.6]

VEDERE ANCHE

fcntl(2) [93.13], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *fd_dup(9)* [99.3.4].

99.3.7 os16: fd_lseek(9)

«

NOME

'**fd_lseek**' - riposizionamento dell'indice di accesso a un descrittore di file

SINTASSI

```
<kernel/fs.h>
off_t fd_lseek (pid_t pid, int fdn, off_t offset, int whence);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`int` ***fdn***	Numero del descrittore a cui si fa riferimento.
`int` ***cmd***	Comando di ***fd_fcntl()***.
`off_t` ***offset***	Scostamento, positivo o negativo, a partire dalla posizione indicata da ***whence***.
`int` ***whence***	Punto di riferimento iniziale a cui applicare lo scostamento.

DESCRIZIONE

La funzione ***fd_lseek()*** consente di riposizionare l'indice di accesso interno al descrittore di file ***fdn***. Per fare questo occorre prima determinare un punto di riferimento, rappresentato dal parametro ***whence***, dove va usata una macro-variabile definita nel file '`lib/unistd.h`'. Può trattarsi dei casi seguenti.

Valore di ***whence***	Significato
`SEEK_SET`	lo scostamento si riferisce all'inizio del file.
`SEEK_CUR`	lo scostamento si riferisce alla posizione che ha già l'indice interno al file.
`SEEK_END`	lo scostamento si riferisce alla fine del file.

Lo scostamento indicato dal parametro ***offset*** si applica a partire dalla posizione a cui si riferisce ***whence***, pertanto può avere segno positivo o negativo, ma in ogni caso non è possibile collocare l'indice prima dell'inizio del file.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_LSEEK`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***fd_lseek()*** è *lseek(2)* [93.24].

VALORE RESTITUITO

Se l'operazione avviene con successo, la funzione restituisce il valore dell'indice riposizionato, preso come scostamento a partire dall'inizio del file. In caso di errore, restituisce invece il valore −1, aggiornando di conseguenza anche la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il numero del descrittore di file non è valido.
`EINVAL`	Il valore di ***whence*** non è contemplato, oppure la combinazione tra ***whence*** e ***offset*** non è valida.

FILE SORGENTI

'`lib/unistd/lseek.c`' [101.17.27]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/fd_lseek.c`' [100.4.7]

VEDERE ANCHE

lseek(2) [93.24], *sysroutine(9)* [99.8.28], *fd_reference(9)* [99.3.10].

99.3.8 os16: fd_open(9)

NOME

'**fd_open**' - apertura di un file puro e semplice oppure di un file di dispositivo

SINTASSI

```
<kernel/fs.h>
int fd_open (pid_t pid, const char *path, int oflags,
             mode_t mode);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`char *`***path***	Il percorso assoluto del file da aprire.
`int` ***oflags***	Opzioni di apertura.
`mode_t` ***mode***	Tipo di file e modalità dei permessi, nel caso il file debba essere creato contestualmente.

DESCRIZIONE

La funzione ***fd_open()*** apre un file, indicato attraverso il percorso ***path***, in base alle opzioni rappresentate dagli indicatori ***oflags***. A seconda del tipo di indicatori specificati, il parametro ***mode*** potrebbe essere preso in considerazione.

Quando la funzione porta a termine correttamente il proprio compito, restituisce il numero del descrittore del file associato, il quale è sempre quello di valore più basso disponibile per il processo elaborativo a cui ci si riferisce.

Il parametro ***oflags*** richiede necessariamente la specificazione della modalità di accesso, attraverso la combinazione appropriata dei valori: '**O_RDONLY**', '**O_WRONLY**', '**O_RDWR**'. Inoltre, si possono combinare altri indicatori: '**O_CREAT**', '**O_TRUNC**', '**O_APPEND**'.

Opzione	Descrizione
`O_RDONLY`	Richiede un accesso in lettura.
`O_WRONLY`	Richiede un accesso in scrittura.
`O_RDWR` `O_RDONLY|O_WRONLY`	Richiede un accesso in lettura e scrittura (la combinazione di '**`R_RDONLY`**' e di '**`O_WRONLY`**' è equivalente all'uso di '**`O_RDWR`**').
`O_CREAT`	Richiede di creare contestualmente il file, ma in tal caso va usato anche il parametro ***mode***.
`O_TRUNC`	Se file da aprire esiste già, richiede che questo sia ridotto preventivamente a un file vuoto.
`O_APPEND`	Fa in modo che le operazioni di scrittura avvengano sempre partendo dalla fine del file.

Quando si utilizza l'opzione ***O_CREAT***, è necessario stabilire la modalità dei permessi, attraverso la combinazione di macro-variabili appropriate, come elencato nella tabella successiva. Tale combinazione va fatta con l'uso dell'operatore OR binario; per esempio: '**`S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH`**'. Va osservato che os16 non gestisce i gruppi di utenti, pertanto, la definizione dei permessi relativi agli utenti appartenenti al gruppo proprietario di un file, non ha poi effetti pratici nel controllo degli accessi per tale tipo di contesto.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXU`	00700_8	Lettura, scrittura ed esecuzione per l'utente proprietario.
`S_IRUSR`	00400_8	Lettura per l'utente proprietario.
`S_IWUSR`	00200_8	Scrittura per l'utente proprietario.
`S_IXUSR`	00100_8	Esecuzione per l'utente proprietario.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXG`	00070_8	Lettura, scrittura ed esecuzione per il gruppo.
`S_IRGRP`	00040_8	Lettura per il gruppo.
`S_IWGRP`	00020_8	Scrittura per il gruppo.
`S_IXGRP`	00010_8	Esecuzione per il gruppo.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXO`	00007_8	Lettura, scrittura ed esecuzione per gli altri utenti.
`S_IROTH`	00004_8	Lettura per gli altri utenti.
`S_IWOTH`	00002_8	Scrittura per gli altri utenti.
`S_IXOTH`	00001_8	Esecuzione per gli altri utenti.

Questa funzione viene usata principalmente da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_OPEN`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***fd_open()*** è *open(2)* [93.28].

VALORE RESTITUITO

La funzione restituisce il numero del descrittore del file aperto, se l'operazione ha avuto successo, altrimenti dà semplicemente -1, impostando di conseguenza il valore della variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
`EINVAL`	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
`EPERM`	Operazione non consentita.
`EEXIST`	Il file da creare esiste già.
`EACCES`	Accesso non consentito.
`ENOENT`	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
`EROFS`	Avendo richiesto un accesso in scrittura, si ottiene che il file system che lo contiene consente soltanto un accesso in lettura.
`ENOTDIR`	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
`ENFILE`	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
`EMFILE`	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

'`lib/fcntl/open.c`' [101.4.3]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/fd_open.c`' [100.4.8]

VEDERE ANCHE

open(2) [93.28], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_inode(9)* [99.3.36], *path_full(9)* [99.3.35], *path_inode_link(9)* [99.3.37], *inode_truncate(9)* [99.3.28], *inode_check(9)* [99.3.16], *file_reference(9)* [99.3.13], *fd_reference(9)* [99.3.10].

99.3.9 os16: fd_read(9)

NOME

'**`fd_read`**' - lettura di descrittore di file

SINTASSI

```
<kernel/fs.h>
ssize_t fd_read (pid_t pid, int fdn, void *buffer,
                 size_t count,
                 int *eof);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`int` ***fdn***	Il numero del descrittore di file.
`void` *****buffer***	Area di memoria in cui scrivere ciò che viene letto dal descrittore di file.
`size_t` ***count***	Quantità di byte da leggere.
`int` *****eof***	Puntatore a una variabile in cui annotare, eventualmente, il raggiungimento della fine del file.

DESCRIZIONE

La funzione ***fd_read()*** cerca di leggere il file rappresentato dal descrittore ***fdn***, partendo dalla posizione in cui si trova l'indice interno di accesso, per un massimo di ***count*** byte, collocando i dati letti in memoria a partire dal puntatore ***buffer***. L'indice interno al file viene fatto avanzare della quantità di byte letti effettivamente, se invece si incontra la fine del file, viene aggiornata la variabile ****eof***.

La funzione può leggere file normali, file di dispositivo e directory, trattandole però come se fossero dei file puri e semplici. Gli altri tipi di file non sono gestiti da os16.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_READ`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***fd_read()*** è *read(2)* [93.29].

VALORE RESTITUITO

La funzione restituisce la quantità di byte letti effettivamente, oppure zero se è stata raggiunta la fine del file e non si può proseguire oltre. Va osservato che la lettura effettiva di una quantità inferiore di byte rispetto a quanto richiesto non costituisce un errore: in quel caso i byte mancanti vanno richiesti eventualmente con successive operazioni di lettura. In caso di errore, la funzione restituisce il valore −1, aggiornando contestualmente la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il numero del descrittore di file non è valido.
`EINVAL`	Il file non è aperto in lettura.
`E_FILE_TYPE_UNSUPPORTED`	Il file è di tipo non gestibile con os16.

FILE SORGENTI

'`lib/unistd/read.c`' [101.17.28]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/fd_read.c`' [100.4.9]

VEDERE ANCHE

read(2) [93.29], *sysroutine(9)* [99.8.28], *fd_reference(9)* [99.3.10], *dev_io(9)* [99.1.1], *inode_file_read(9)* [99.3.18].

99.3.10 os16: fd_reference(9)

«

NOME

'**`fd_reference`**' - riferimento a un elemento della tabella dei descrittori

SINTASSI

```
<kernel/fs.h>
fd_t *fd_reference (pid_t pid, int *fdn);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`int` ***fdn***	Il numero del descrittore di file.

DESCRIZIONE

La funzione ***fd_reference()*** restituisce il puntatore all'elemento della tabella dei descrittori, corrispondente al processo e al numero di descrittore specificati. Se però viene fornito un numero di descrittore negativo, si ottiene il puntatore al primo elemento che risulta libero nella tabella.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella dei descrittori, oppure il puntatore nullo in caso di errore, ma **senza aggiornare** la variabile ***errno*** del kernel. Infatti, l'unico errore che può verificarsi consiste nel non poter trovare il descrittore richiesto.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/fd_reference.c`' [100.4.10]

VEDERE ANCHE

file_reference(9) [99.3.13], *inode_reference(9)* [99.3.25], *sb_reference(9)* [99.3.47], *proc_reference(9)* [99.8.7].

99.3.11 os16: fd_stat(9)

Vedere *stat(9)* [99.3.50].

99.3.12 os16: fd_write(9)

NOME

'**`fd_write`**' - scrittura di un descrittore di file

SINTASSI

```
<kernel/fs.h>
ssize_t fd_write (pid_t pid, int fdn, const void *buffer,
                  size_t count);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`int` ***fdn***	Il numero del descrittore di file.
`const void *`***buffer***	Area di memoria da cui attingere i dati da scrivere nel descrittore di file.
`size_t` ***count***	Quantità di byte da scrivere.

DESCRIZIONE

La funzione ***fd_write()*** consente di scrivere fino a un massimo di ***count*** byte, tratti dall'area di memoria che inizia all'indirizzo ***buffer***, presso il file rappresentato dal descrittore ***fdn***, del processo ***pid***. La scrittura avviene a partire dalla posizione in cui si trova l'indice interno.

Questa funzione viene usata principalmente da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_WRITE`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***fd_write()*** è *write(2)* [93.44].

VALORE RESTITUITO

La funzione restituisce la quantità di byte scritti effettivamente e in tal caso è possibile anche ottenere una quantità pari a zero. Se si verifica invece un errore, la funzione restituisce -1 e aggiorna la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EBADF`	Il numero del descrittore di file non è valido.
`EINVAL`	Il file non è aperto in scrittura.
`EISDIR`	Il file è una directory.
`E_FILE_TYPE_UNSUPPORTED`	Il file è di tipo non gestibile con os16.
`EIO`	Errore di input-output.

FILE SORGENTI

‘`lib/unistd/write.c`’ [101.17.36]

‘`kernel/proc.h`’ [100.9]

‘`kernel/proc/_isr.s`’ [100.9.1]

‘`kernel/proc/sysroutine.c`’ [100.9.30]

‘`kernel/fs.h`’ [100.4]

‘`kernel/fs/fd_write.c`’ [100.4.12]

VEDERE ANCHE

write(2) [93.44], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *dev_io(9)* [99.1.1], *inode_file_write(9)* [99.3.19].

99.3.13 os16: file_reference(9)

NOME

‘**`file_reference`**’ - riferimento a un elemento della tabella dei file di sistema

SINTASSI

```
<kernel/fs.h>
file_t *file_reference (int fno);
```

ARGOMENTI

Argomento	Descrizione
`int` ***fno***	Il numero della voce della tabella dei file, a partire da zero.

DESCRIZIONE

La funzione ***file_reference()*** restituisce il puntatore all’elemento della tabella dei file di sistema, corrispondente al numero indicato come argomento. Se però tale numero fosse negativo, viene restituito il puntatore al primo elemento libero.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella dei file di sistema, oppure il puntatore nullo in caso di errore, ma **senza aggiornare** la variabile ***errno*** del kernel. Infatti, l’unico errore che può verificarsi consiste nel non poter trovare la voce richiesta.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/file_table.c`' [100.4.15]

'`kernel/fs/file_reference.c`' [100.4.13]

VEDERE ANCHE

fd_reference(9) [99.3.10], *inode_reference(9)* [99.3.25], *sb_reference(9)* [99.3.47], *proc_reference(9)* [99.8.7].

99.3.14 os16: file_stdio_dev_make(9)

«

NOME

'**`file_stdio_dev_make`**' - creazione di una voce relativa a un dispositivo di input-output standard, nella tabella dei file di sistema

SINTASSI

```
<kernel/fs.h>
file_t *file_stdio_dev_make (dev_t device, mode_t mode,
                             int oflags);
```

ARGOMENTI

Argomento	Descrizione
`dev_t` ***device***	Il numero del dispositivo da usare per l'input o l'output.
`mode_t` ***mode***	Tipo di file di dispositivo: è praticamente obbligatorio l'uso di '**`S_IFCHR`**'.
`int` ***oflags***	Modalità di accesso: '**`O_RDONLY`**' oppure '**`O_WRONLY`**'.

DESCRIZIONE

La funzione ***file_stdio_dev_make()*** produce una voce nella tabella dei file di sistema, relativa a un dispositivo di input-output, da usare come flusso standard. In altri termini, serve per creare le voci della tabella dei file, relative a standard input, standard output e standard error.

Per ottenere questo risultato occorre coinvolgere anche la funzione *inode_stdio_dev_make(9)* [99.3.27], la quale si occupa di predisporre un inode, privo però di un collegamento a un file vero e proprio.

Questa funzione viene usata esclusivamente da *proc_sys_exec(9)* [99.8.20], per attribuire standard input, standard output e standard error, che non fossero già disponibili.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella dei file di sistema, oppure il puntatore nullo in caso di errore, aggiornando la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`ENFILE`	Non è possibile allocare un altro file di sistema.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/file_stdio_dev_make.c`' [100.4.14]

VEDERE ANCHE

proc_sys_exec(9) [99.8.20], *inode_stdio_dev_make(9)* [99.3.27], *file_reference(9)* [99.3.13], *inode_put(9)* [99.3.24].

99.3.15 os16: inode_alloc(9)

«

NOME

'**`inode_alloc`**' - allocazione di un inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_alloc (dev_t device, mode_t mode, uid_t uid);
```

ARGOMENTI

Argomento	Descrizione
`dev_t` ***device***	Il numero del dispositivo in cui si trova il file system dove allocare l'inode.
`mode_t` ***mode***	Tipo di file e modalità dei permessi da associare all'inode.
`uid_t` ***uid***	Proprietà dell'inode.

DESCRIZIONE

La funzione ***inode_alloc()*** cerca un inode libero nel file system del dispositivo indicato, quindi lo alloca (lo segna come utilizzato) e lo modifica aggiornando il tipo e la modalità dei permessi, oltre al proprietario del file. Se la funzione riesce nel suo intento, restituisce il puntatore all'inode in memoria, il quale rimane così aperto e disponibile per ulteriori elaborazioni.

Questa funzione viene usata esclusivamente da *path_inode_link(9)* [99.3.37], per la creazione di un nuovo file.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella degli inode di sistema, oppure il puntatore nullo in caso di errore, aggiornando la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il valore fornito per il parametro ***mode*** non è ammissibile.
`ENODEV`	Il dispositivo non corrisponde ad alcuna voce della tabella dei super blocchi; per esempio, il file system cercato potrebbe non essere ancora stato innestato.
`ENOSPC`	Non è possibile allocare l'inode, per mancanza di spazio.
`ENFILE`	Non c'è spazio nella tabella degli inode.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_alloc.c`' [100.4.16]

VEDERE ANCHE

path_inode_link(9) [99.3.37], *sb_reference(9)* [99.3.47], *inode_get(9)* [99.3.23], *inode_put(9)* [99.3.24], *inode_truncate(9)* [99.3.28], *inode_save(9)* [99.3.26].

99.3.16 os16: inode_check(9)

«

NOME

'**`inode_check`**' - verifica delle caratteristiche di un inode

SINTASSI

```
<kernel/fs.h>
int inode_check (inode_t *inode, mode_t type, int perm,
                 uid_t uid);
```

ARGOMENTI

Argomento	Descrizione
`inode_t *inode`	Puntatore a un elemento della tabella degli inode.
`mode_t type`	Tipo di file desiderato. Può trattarsi di '**S_IFBLK**', '**S_IFCHR**', '**S_IFIFO**', '**S_IFREG**', '**S_IFDIR**', '**S_IFLNK**', '**S_IFSOCK**', come dichiarato nel file '`lib/sys/stat.h`', tuttavia os16 gestisce solo file di dispositivo, file normali e directory.
`int perm`	Permessi richiesti dall'utente ***uid***, rappresentati nei tre bit meno significativi.
`uid_t uid`	Utente nei confronti del quale vanno verificati i permessi di accesso.

DESCRIZIONE

La funzione ***inode_check()*** verifica che l'inode indicato sia di un certo tipo e abbia i permessi di accesso necessari a un certo utente. Tali permessi vanno rappresentati utilizzando solo gli ultimi tre bit (4 = lettura, 2 = scrittura, 1 = esecuzione o attraversamento) e si riferiscono alla richiesta di accesso all'inode, da parte dell'utente ***uid***, tenendo conto del complesso dei permessi che lo riguardano.

Nel parametro ***type*** è ammessa la sovrapposizione di più tipi validi.

Questa funzione viene usata in varie situazioni, internamente al kernel, per verificare il tipo o l'accessibilità di un file.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Le caratteristiche dell'inode sono compatibili con quanto richiesto.
−1	Le caratteristiche dell'inode non sono compatibili, oppure si è verificato un errore. In ogni caso si ottiene l'aggiornamento della variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
EINVAL	Il valore di ***inode*** corrisponde a un puntatore nullo.
E_FILE_TYPE	Il tipo di file dell'inode non corrisponde a quanto richiesto.
EACCES	I permessi di accesso non sono compatibili con la richiesta.

FILE SORGENTI

'kernel/fs.h' [100.4]

'kernel/fs/inode_check.c' [100.4.17]

99.3.17 os16: inode_dir_empty(9)

«

NOME

'**inode_dir_empty**' - verifica della presenza di contenuti in una directory

SINTASSI

```
<kernel/fs.h>
int inode_dir_empty (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
inode_t **inode*	Puntatore a un elemento della tabella degli inode.

DESCRIZIONE

La funzione ***inode_dir_empty()*** verifica che la directory, a cui si riferisce l'inode a cui punta ***inode***, sia vuota.

VALORE RESTITUITO

Valore	Significato del valore restituito
1	*Vero*: si tratta di una directory vuota.
0	*Falso*: se è effettivamente una directory, questa non è vuota, altrimenti non è nemmeno una directory.

ERRORI

Dal momento che un risultato *Falso* non rappresenta necessariamente un errore, per verificare il contenuto della variabile ***errno***, prima dell'uso della funzione occorre azzerarla.

Valore di ***errno***	Significato
`EINVAL`	L'inode non riguarda una directory.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_dir_empty.c`' [100.4.18]

VEDERE ANCHE

inode_file_read(9) [99.3.18].

99.3.18 os16: inode_file_read(9)

NOME

'**`inode_file_read`**' - lettura di un file rappresentato da un inode

SINTASSI

```
<kernel/fs.h>
ssize_t inode_file_read (inode_t *inode, off_t offset,
                         void *buffer, size_t count,
                         int *eof);
```

ARGOMENTI

Argomento	Descrizione
`inode_t *`***inode***	Puntatore a un elemento della tabella degli inode, che rappresenta il file da leggere.
`off_t` ***offset***	Posizione, riferita all'inizio del file, a partire dalla quale eseguire la lettura.
`void *`***buffer***	Puntatore all'area di memoria in cui scrivere ciò che si ottiene dalla lettura del file.
`size_t` ***count***	Quantità massima di byte da leggere.
`int *`***eof***	Puntatore a un indicatore di fine file, da aggiornare (purché sia un puntatore valido) in base all'esito della lettura.

DESCRIZIONE

La funzione ***inode_file_read()*** legge il contenuto del file a cui si riferisce l'inode ***inode*** e se il puntatore ***eof*** è valido, aggiorna anche la variabile ****eof***.

Questa funzione si avvale a sua volta di *inode_fzones_read(9)* [99.3.21], per accedere ai contenuti del file, suddivisi in zone, secondo l'organizzazione del file system Minix 1.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letti e resi effettivamente disponibili a partire da ciò a cui punta ***buffer***. Se la variabile ***var*** è un puntatore valido, aggiorna anche il suo valore, azzerandolo se la lettura avviene in una posizione interna al file, oppure impostandolo a uno se la lettura richiesta è oltre la fine del file. Se invece si tenta una lettura con un valore di ***offset*** negativo, o specificando il puntatore nullo al posto dell'inode, la funzione restituisce −1 e aggiorna la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	L'inode non è valido, oppure il valore di ***offset*** è negativo.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_file_read.c`' [100.4.19]

VEDERE ANCHE

inode_fzones_read(9) [99.3.21].

99.3.19 os16: inode_file_write(9)

«

NOME

'**`inode_file_write`**' - scrittura di un file rappresentato da un inode

SINTASSI

```
<kernel/fs.h>
ssize_t inode_file_write (inode_t *inode, off_t offset,
                          void *buffer, size_t count);
```

ARGOMENTI

Argomento	Descrizione
`inode_t *`***inode***	Puntatore a un elemento della tabella degli inode, che rappresenta il file da scrivere.
`off_t` ***offset***	Posizione, riferita all'inizio del file, a partire dalla quale eseguire la scrittura.
`void *`***buffer***	Puntatore all'area di memoria da cui trarre i dati da scrivere nel file.
`size_t` ***count***	Quantità massima di byte da scrivere.

DESCRIZIONE

La funzione ***inode_file_write()*** scrive nel file rappresentato da ***inode***, a partire dalla posizione ***offset*** (purché non sia un valore negativo), la quantità massima di byte indicati con ***count***, ciò che si trova in memoria a partire da ***buffer***.

Questa funzione si avvale a sua volta di *inode_fzones_read(9)* [99.3.21], per accedere ai contenuti del file, suddivisi in zone, secondo l'organizzazione del file system Minix 1, e di *zone_write(9)* [99.3.53], per la riscrittura delle zone relative.

Per os16, le operazioni di scrittura nel file system sono sincrone, senza alcun trattenimento in memoria (ovvero senza *cache*).

VALORE RESTITUITO

La funzione restituisce la quantità di byte scritti. La scrittura può avvenire oltre la fine del file, anche in modo discontinuo; tuttavia, non è ammissibile un valore di ***offset*** negativo.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	L'inode non è valido, oppure il valore di ***offset*** è negativo.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_file_write.c`' [100.4.20]

VEDERE ANCHE

inode_fzones_read(9) [99.3.21], *zone_write(9)* [99.3.53].

99.3.20 os16: inode_free(9)

NOME

'**inode_free**' - deallocazione di un inode

SINTASSI

```
<kernel/fs.h>
int inode_free (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
`inode_t *`***inode***	Puntatore a un elemento della tabella degli inode.

DESCRIZIONE

La funzione ***inode_free()*** libera l'inode specificato attraverso il puntatore ***inode***, rispetto al proprio super blocco. L'operazione comporta semplicemente il fatto di indicare questo inode come libero, senza controlli per verificare se effettivamente non esistono più collegamenti nel file system che lo riguardano.

Questa funzione viene usata esclusivamente da *inode_put(9)* [99.3.24], per completare la cancellazione di un inode che non ha più collegamenti nel file system, nel momento in cui non vi si fa più riferimento nel sistema in funzione.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dalla variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	L'inode non è valido.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_free.c`' [100.4.21]

VEDERE ANCHE

inode_save(9) [99.3.26], *inode_alloc(9)* [99.3.15].

99.3.21 os16: inode_fzones_read(9)

NOME

'**inode_fzones_read**', '**inode_fzones_write**' - lettura e scrittura di zone relative al contenuto di un file

SINTASSI

```
<kernel/fs.h>
blkcnt_t inode_fzones_read  (inode_t *inode, zno_t zone_start,
                             void *buffer, blkcnt_t blkcnt);
blkcnt_t inode_fzones_write (inode_t *inode, zno_t zone_start,
                             void *buffer, blkcnt_t blkcnt);
```

ARGOMENTI

Argomento	Descrizione
`inode_t` ***inode**	Puntatore a un elemento della tabella degli inode, con cui si individua il file da cui leggere o in cui scrivere.
`zno_t` ***zone_start***	Il numero di zona, relativo al file, a partire dalla quale iniziare la lettura o la scrittura.
`void` ****buffer***	Il puntatore a un'area di memoria tampone, da usare per depositare i dati letti o per trarre i dati da scrivere.
`blkcnt_t` ***blkcnt***	La quantità di zone da leggere o scrivere.

DESCRIZIONE

Le funzioni ***inode_fzones_read()*** e ***inode_fzones_write()***, consentono di leggere e di scrivere un file, a zone intere (la zona è un multiplo del blocco, secondo la filosofia del file system Minix 1).

Questa funzione vengono usate soltanto da *inode_file_read(9)* [99.3.18] e *inode_file_write(9)* [99.3.19], con le quali l'accesso ai file si semplifica a livello di byte.

VALORE RESTITUITO

Le due funzioni restituiscono la quantità di zone lette o scritte effettivamente. Una quantità pari a zero potrebbe eventualmente rappresentare un errore, ma solo in alcuni casi. Per poterlo verificare, occorre azzerare la variabile ***errno*** prima di chiamare le funzioni, riservandosi di verificarne successivamente il valore.

ERRORI

Valore di ***errno***	Significato
`EIO`	L'accesso alla zona richiesta non è potuto avvenire.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_fzones_read.c`' [100.4.22]

'`kernel/fs/inode_fzones_write.c`' [100.4.23]

VEDERE ANCHE

inode_file_read(9) [99.3.18], *inode_file_write(9)* [99.3.19], *zone_read(9)* [99.3.53], *zone_write(9)* [99.3.53].

99.3.22 os16: inode_fzones_write(9)

Vedere *inode_fzones_read(9)* [99.3.21].

99.3.23 os16: inode_get(9)

NOME

'**`inode_get`**' - caricamento di un inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_get (dev_t device, ino_t ino);
```

ARGOMENTI

Argomento	Descrizione
`dev_t` ***device***	Dispositivo riferito a un'unità di memorizzazione, dove cercare l'inode numero ***ino***.
`ino_t` ***ino***	Numero di inode, relativo al file system contenuto nell'unità ***device***.

DESCRIZIONE

La funzione ***inode_get()*** consente di «aprire» un inode, fornendo il numero del dispositivo corrispondente all'unità di memorizzazione e il numero dell'inode del file system in essa contenuto. L'inode in questione potrebbe essere già stato aperto e quindi già disponibile in memoria nella tabella degli inode; in tal caso, la

funzione si limita a incrementare il contatore dei riferimenti a tale inode, da parte del sistema in funzione, restituendo il puntatore all'elemento della tabella che lo contiene gia. Se invece l'inode non è ancora presente nella tabella rispettiva, la funzione deve provvedere a caricarlo.

Se si richiede un inode non ancora disponibile, contenuto in un'unità di cui non è ancora stato caricato il super blocco nella tabella rispettiva, la funzione deve provvedere anche a questo procedimento.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella degli inode che rappresenta l'inode aperto. Se però si presenta un problema, restituisce il puntatore nullo e aggiorna la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EUNKNOWN`	Si tratta di un problema imprevisto e non meglio identificabile.
`ENFILE`	La tabella degli inode è già occupata completamente e non è possibile aprirne altri.
`ENODEV`	Il dispositivo richiesto non è valido.
`ENOENT`	Il numero di inode richiesto non esiste.
`EIO`	Errore nella lettura del file system.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_get.c`' [100.4.24]

VEDERE ANCHE

offsetof(3) [94.75], *inode_put(9)* [99.3.24], *inode_reference(9)* [99.3.25], *sb_reference(9)* [99.3.47], *sb_inode_status(9)* [99.3.45], *dev_io(9)* [99.1.1].

99.3.24 os16: inode_put(9)

NOME

'**`inode_put`**' - rilascio di un inode

SINTASSI

```
<kernel/fs.h>
int inode_put (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
`inode_t *inode`	Puntatore a un elemento della tabella di inode.

DESCRIZIONE

La funzione ***inode_put()*** «chiude» un inode, riducendo il contatore degli accessi allo stesso. Tuttavia, se questo contatore, dopo il decremento, raggiunge lo zero, è necessario verificare se nel frattempo anche i collegamenti del file system si sono azzerati, perché in tal caso occorre anche rimuovere l'inode, nel senso di segnalarlo come libero per la creazione di un nuovo file. In ogni caso, le informazioni aggiornate dell'inode, ancora allocato o liberato, vengono memorizzate nel file system.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dalla variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il puntatore ***inode*** non è valido.
`EUNKNOWN`	Si tratta di un problema imprevisto e non meglio identificabile.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_put.c`' [100.4.25]

VEDERE ANCHE

inode_truncate(9) [99.3.28], *inode_free(9)* [99.3.20], *inode_save(9)* [99.3.26].

99.3.25 os16: inode_reference(9)

NOME

'**inode_reference**' - riferimento a un elemento della tabella di inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_reference (dev_t device, ino_t ino);
```

ARGOMENTI

Argomento	Descrizione
`dev_t` ***device***	Dispositivo riferito a un'unità di memorizzazione, dove cercare l'inode numero ***ino***.
`ino_t` ***ino***	Numero di inode, relativo al file system contenuto nell'unità ***device***.

DESCRIZIONE

La funzione ***inode_reference()*** cerca nella tabella degli inode la voce corrispondente ai dati forniti come argomenti, ovvero quella dell'inode numero ***ino*** del file system contenuto nel dispositivo ***device***, restituendo il puntatore alla voce corrispondente. Tuttavia ci sono dei casi particolari:

- se il numero del dispositivo e quello dell'inode sono entrambi zero, viene restituito il puntatore all'inizio della tabella, ovvero al primo elemento della stessa;
- se il numero del dispositivo e quello dell'inode sono pari a un numero negativo (rispettivamente '**(dev_t) -1**' e '**(ino_t) -1**'), viene restituito il puntatore alla prima voce libera;
- se il numero del dispositivo è pari a zero e il numero dell'inode è pari a uno, si intende ricercare la voce dell'inode della directory radice del file system principale.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella degli inode, se la ricerca si compie con successo. In caso di problemi, invece, la funzione restituisce il puntatore nullo e aggiorna la variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
E_CANNOT_FIND_ROOT_DEVICE	Nella tabella dei super blocchi non è possibile trovare il file system principale.
E_CANNOT_FIND_ROOT_INODE	Nella tabella degli inode non è possibile trovare la directory radice del file system principale.

FILE SORGENTI

'kernel/fs.h' [100.4]

'kernel/fs/inode_reference.c' [100.4.26]

VEDERE ANCHE

sb_reference(9) [99.3.47], *file_reference(9)* [99.3.13], *proc_reference(9)* [99.8.7].

99.3.26 os16: inode_save(9)

«

NOME

'**inode_save**' - memorizzazione dei dati di un inode

SINTASSI

```
<kernel/fs.h>
int inode_save (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
inode_t **inode*	Puntatore a una voce della tabella degli inode.

DESCRIZIONE

La funzione ***inode_save()*** memorizza l'inode a cui si riferisce la voce ***inode***, nel file system, ammesso che si tratti effettivamente di un inode relativo a un file system e che sia stato modificato dopo l'ultima memorizzazione precedente. In questo caso, la funzione, a sua volta, richiede la memorizzazione del super blocco.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dalla variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
`EINVAL`	Il puntatore ***inode*** è nullo.

FILE SORGENTI

‘`kernel/fs.h`’ [100.4]

‘`kernel/fs/inode_save.c`’ [100.4.27]

VEDERE ANCHE

sb_save(9) [99.3.48], *dev_io(9)* [99.1.1].

99.3.27 os16: inode_stdio_dev_make(9)

NOME

‘**`inode_stdio_dev_make`**’ - creazione di una voce relativa a un dispositivo di input-output standard, nella tabella degli inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_stdio_dev_make (dev_t device, mode_t mode);
```

ARGOMENTI

Argomento	Descrizione
`dev_t` ***device***	Il numero del dispositivo da usare per l’input o l’output.
`mode_t` ***mode***	Tipo di file di dispositivo: è praticamente obbligatorio l’uso di ‘**`S_IFCHR`**’.

DESCRIZIONE

La funzione ***inode_stdio_dev_make()*** produce una voce nella tabella degli inode, relativa a un dispositivo di input-output, da usare come flusso standard. In altri termini, serve per creare le voci della tabella degli inode, relative a standard input, standard output e standard error.

Questa funzione viene usata esclusivamente da *file_stdio_dev_make(9)* [99.3.14], per creare una voce da usare come flusso standard di input o di output, nella tabella dei file.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella degli inode, oppure il puntatore nullo in caso di errore, aggiornando la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Gli argomenti della chiamata non sono validi.
`ENFILE`	Non è possibile allocare un altro inode.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_stdio_dev_make.c`' [100.4.28]

VEDERE ANCHE

file_stdio_dev_make(9) [99.3.14], *inode_reference(9)* [99.3.25].

99.3.28 os16: inode_truncate(9)

«

NOME

'**`inode_truncate`**' - troncamento del file a cui si riferisce un inode

SINTASSI

```
<kernel/fs.h>
int inode_truncate (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
`inode_t *`***inode***	Puntatore a una voce della tabella di inode.

DESCRIZIONE

La funzione ***inode_truncate()*** richiede che il puntatore ***inode*** si riferisca a una voce della tabella degli inode, relativa a un file contenuto in un file system. Lo scopo della funzione è annullare il contenuto di tale file, trasformandolo in un file vuoto.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dalla variabile ***errno*** del kernel.

ERRORI

Allo stato attuale dello sviluppo della funzione, non ci sono controlli e non sono previsti errori.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_truncate.c`' [100.4.30]

VEDERE ANCHE

zone_free(9) [99.3.51], *sb_save(9)* [99.3.48], *inode_save(9)* [99.3.26].

99.3.29 os16: inode_zone(9)

NOME

'**`inode_zone`**' - traduzione del numero di zona relativo in un numero di zona assoluto

SINTASSI

```
<kernel/fs.h>
zno_t inode_zone (inode_t *inode, zno_t fzone, int write);
```

ARGOMENTI

Argomento	Descrizione
`inode_t` ***inode***	Puntatore a una voce della tabella di inode.
`zno_t` ***fzone***	Numero di zona relativo al file dell'inode preso in considerazione.
`int` ***write***	Valore da intendersi come *Vero* o *Falso*, con cui consentire o meno la creazione al volo di una zona mancante.

DESCRIZIONE

La funzione ***inode_zone()*** serve a tradurre il numero di una zona, inteso relativamente a un file, nel numero assoluto relativamente al file system in cui si trova.

Tuttavia, un file può essere memorizzato effettivamente in modo discontinuo, ovvero con zone inesistenti nella sua parte centrale. Il contenuto di un file che non dispone effettivamente di zone allocate, corrisponde a un contenuto nullo dal punto di vista binario (zero binario), ma per la funzione, una zona assente comporta la restituzione di un valore nullo, perché nel file system non c'è. Pertanto, se l'argomento corrispondente al parametro ***write*** contiene un valore diverso da zero, la funzione che non trova una zona, la alloca e quindi ne restituisce il numero.

VALORE RESTITUITO

La funzione restituisce il numero della zona che nel file system corrisponde a quella relativa richiesta per un certo file. Nel caso la zona non esista, perché non allocata, restituisce zero. Tuttavia, la zona zero di un file system Minix 1 esiste, ma contiene sostanzialmente le informazioni amministrative del super blocco, pertanto non può essere una traduzione valida di una zona di un file.

ERRORI

La funzione non prevede il verificarsi di errori.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/inode_zone.c`' [100.4.31]

VEDERE ANCHE

memset(3) [94.72], *zone_alloc(9)* [99.3.51], *zone_read(9)* [99.3.53], *zone_write(9)* [99.3.53].

99.3.30 os16: path_chdir(9)

NOME

'**`path_chdir`**' - cambiamento della directory corrente

SINTASSI

```
<kernel/fs.h>
int path_chdir (pid_t pid, const char *path);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`const char *`***path***	Il percorso della nuova directory corrente, riferito alla directory corrente del processo ***pid***.

DESCRIZIONE

La funzione ***path_chdir()*** cambia la directory corrente del processo ***pid***, in modo che quella nuova corrisponda al percorso annotato nella stringa ***path***.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_CHDIR`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***path_chdir()*** è *chdir(2)* [93.3].

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Argomento non valido.
`EACCES`	Accesso negato.
`ENOTDIR`	Uno dei componenti del percorso non è una directory.
`ENOENT`	Uno dei componenti del percorso non esiste.

FILE SORGENTI

'`lib/unistd/chdir.c`' [101.17.3]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs/path_chdir.c`' [100.4.32]

VEDERE ANCHE

chdir(2) [93.3], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_full(9)* [99.3.35], *path_inode(9)* [99.3.36], *inode_put(9)* [99.3.24].

99.3.31 os16: path_chmod(9)

«

NOME

'**path_chmod**' - cambiamento della modalità dei permessi di un file

SINTASSI

```
<kernel/fs.h>
int path_chmod (pid_t pid, const char *path, mode_t mode);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`const char *`***path***	Il percorso del file su cui intervenire.
`mode_t` ***mode***	La modalità dei permessi da applicare (contano solo i 12 bit meno significativi).

DESCRIZIONE

La funzione ***path_chmod()*** modifica la modalità dei permessi di accesso del file indicato, tramite il suo percorso, relativo eventualmente alla directory corrente del processo ***pid***.

Tradizionalmente, i permessi si scrivono attraverso un numero in base otto; in alternativa, si possono usare convenientemente della macro-variabili, dichiarate nel file '`lib/sys/stat.h`', combinate assieme con l'operatore binario OR.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXU`	00700_8	Lettura, scrittura ed esecuzione per l'utente proprietario.
`S_IRUSR`	00400_8	Lettura per l'utente proprietario.
`S_IWUSR`	00200_8	Scrittura per l'utente proprietario.
`S_IXUSR`	00100_8	Esecuzione per l'utente proprietario.
`S_IRWXG`	00070_8	Lettura, scrittura ed esecuzione per il gruppo.
`S_IRGRP`	00040_8	Lettura per il gruppo.
`S_IWGRP`	00020_8	Scrittura per il gruppo.
`S_IXGRP`	00010_8	Esecuzione per il gruppo.
`S_IRWXO`	00007_8	Lettura, scrittura ed esecuzione per gli altri utenti.
`S_IROTH`	00004_8	Lettura per gli altri utenti.
`S_IWOTH`	00002_8	Scrittura per gli altri utenti.
`S_IXOTH`	00001_8	Esecuzione per gli altri utenti.

os16 non considera i permessi SUID (*Set user id*), SGID (*Set group id*) e Sticky, che nella tabella non sono stati nemmeno annotati; inoltre, non tiene in considerazione i permessi legati al gruppo, perché non tiene traccia dei gruppi.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
`EACCES`	Permesso negato.
`EBADF`	Il descrittore del file richiesto non è valido.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_CHMOD`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***path_chmod()*** è *chmod(2)* [93.4].

FILE SORGENTI

'`lib/sys/stat/chmod.c`' [101.13.1]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_chmod.c`' [100.4.33]

VEDERE ANCHE

chmod(2) [93.4], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_inode(9)* [99.3.36].

99.3.32 os16: path_chown(9)

«

NOME

'**`path_chown`**' - cambiamento della proprietà di un file

SINTASSI

```
<kernel/fs.h>
int path_chown (pid_t pid, const char *path, uid_t uid,
                gid_t gid);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`const char *`***path***	Il percorso del file su cui intervenire.
`uid_t` ***uid***	Utente a cui attribuire la proprietà del file.
`gid_t` ***gid***	Gruppo a cui associare il file.

DESCRIZIONE

La funzione ***path_chown()*** modifica la proprietà di un file, fornendo il numero UID e il numero GID. Il file viene indicato attraverso il percorso scritto in una stringa, relativo alla directory corrente del processo ***pid***.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_CHOWN`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***path_chown()*** è *chown(2)* [93.5].

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Argomento non valido.
`EPERM`	Permessi insufficienti per eseguire l'operazione.
`ENOTDIR`	Uno dei componenti del percorso non è una directory.
`ENOENT`	Uno dei componenti del percorso non esiste.
`EBADF`	Il descrittore del file non è valido.

DIFETTI

Benché sia consentito di attribuire il numero del gruppo, os16 non valuta i permessi di accesso ai file, relativi a questi.

FILE SORGENTI

'`lib/unistd/chown.c`' [101.17.4]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_chown.c`' [100.4.34]

VEDERE ANCHE

chown(2) [93.5], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_inode(9)* [99.3.36], *inode_save(9)* [99.3.26], *inode_put(9)* [99.3.24].

99.3.33 os16: path_device(9)

«

NOME

'**`path_device`**' - conversione di un file di dispositivo nel numero corrispondente

SINTASSI

```
<kernel/fs.h>
dev_t path_device (pid_t pid, const char *path);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Processo elaborativo per conto del quale si agisce.
`const char *`***path***	Il percorso del file di dispositivo.

DESCRIZIONE

La funzione ***path_device()*** consente di trarre il numero complessivo di un dispositivo, a partire da un file di dispositivo.

Questa funzione viene usata soltanto da *path_mount(9)* [99.8.28].

VALORE RESTITUITO

La funzione restituisce il numero del dispositivo corrispondente al file indicato, oppure il valore -1, in caso di errore, aggiornando la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`ENODEV`	Il file richiesto non è un file di dispositivo.
`ENOENT`	Il file richiesto non esiste.
`EACCES`	Il file richiesto non è accessibile secondo i privilegi del processo ***pid***.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_device.c`' [100.4.35]

VEDERE ANCHE

proc_reference(9) [99.8.7], *path_inode(9)* [99.3.36], *inode_put(9)* [99.3.24].

99.3.34 os16: path_fix(9)

NOME

'**`path_fix`**' - semplificazione di un percorso

SINTASSI

```
<kernel/fs.h>
int path_fix (char *path);
```

ARGOMENTI

Argomento	Descrizione
`char *`***path***	Il percorso da semplificare.

DESCRIZIONE

La funzione ***path_fix()*** legge la stringa del percorso ***path*** e la rielabora, semplificandolo. La semplificazione riguarda l'eliminazione di riferimenti inutili alla directory corrente e di indietreggiamenti. Il percorso può essere assoluto o relativo: la funzione non ne cambia l'origine.

VALORE RESTITUITO

La funzione restituisce sempre zero e non è prevista la manifestazione di errori.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_fix.c`' [100.4.36]

VEDERE ANCHE

strtok(3) [94.120], *strcmp(3)* [94.106], *strcat(3)* [94.104], *strncat(3)* [94.104], *strncpy(3)* [94.108].

99.3.35 os16: path_full(9)

«

NOME

'**`path_full`**' - traduzione di un percorso relativo in un percorso assoluto

SINTASSI

```
<kernel/fs.h>
int path_full (const char *path, const char *path_cwd,
               char *full_path);
```

ARGOMENTI

Argomento	Descrizione
`const char *`***path***	Il percorso relativo alla posizione ***path_cwd***.
`const char *`***path_cwd***	La directory corrente.
`char *`***full_path***	Il luogo in cui scrivere il percorso assoluto.

DESCRIZIONE

La funzione ***path_full()*** ricostruisce un percorso assoluto, mettendolo in memoria a partire da ciò a cui punta ***full_path***.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
`EINVAL`	L'insieme degli argomenti non è valido.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_full.c`' [100.4.37]

VEDERE ANCHE

strtok(3) [94.120], *strcmp(3)* [94.106], *strcat(3)* [94.104], *strncat(3)* [94.104], *strncpy(3)* [94.108], *path_fix(9)* [99.3.34].

99.3.36 os16: path_inode(9)

«

NOME

'**`path_inode`**' - caricamento di un inode, partendo dal percorso del file

SINTASSI

```
<kernel/fs.h>
inode_t *path_inode (pid_t pid, const char *path);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`const char *`***path***	Il percorso del file del quale si vuole ottenere l'inode.

DESCRIZIONE

La funzione ***path_inode()*** carica un inode nella tabella degli inode, oppure lo localizza se questo è già caricato, partendo dal percorso di un file. L'operazione è subordinata all'accessibilità del percorso che conduce al file, nel senso che il processo ***pid*** deve avere il permesso di accesso («x») in tutti gli stadi dello stesso.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella degli inode che contiene le informazioni caricate in memoria sull'inode. Se qualcosa non va, restituisce invece il puntatore nullo, aggiornando di conseguenza il contenuto della variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
`ENOENT`	Uno dei componenti del percorso non esiste.
`ENFILE`	Non è possibile allocare un inode ulteriore, perché la tabella è già occupata completamente.
`EIO`	Error di input-output.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_inode.c`' [100.4.38]

VEDERE ANCHE

proc_reference(9) [99.8.7], *path_full(9)* [99.3.35], *inode_get(9)* [99.3.23], *inode_put(9)* [99.3.24], *inode_check(9)* [99.3.16], *inode_file_read(9)* [99.3.18].

99.3.37 os16: path_inode_link(9)

NOME

'**`path_inode_link`**' - creazione di un collegamento fisico o di un nuovo file

SINTASSI

```
<kernel/fs.h>
inode_t *path_inode_link (pid_t pid, const char *path,
                          inode_t *inode, mode_t mode);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`const char *`***path***	Il percorso del file per il quale si vuole creare il collegamento fisico.
`inode_t *`***inode***	Puntatore a una voce della tabella degli inode, alla quale si vuole collegare il nuovo file.
`mode_t` ***mode***	Nel caso l'inode non sia stato fornito, dovendo creare un nuovo file, questo parametro richiede il tipo e i permessi del file da creare.

DESCRIZIONE

La funzione ***path_inode_link()*** crea un collegamento fisico con il nome fornito in ***path***, riferito all'inode a cui punta ***inode***. Tuttavia, l'argomento corrispondente al parametro ***inode*** può essere un puntatore nullo, e in tal caso viene creato un file vuoto, allocando contestualmente un nuovo inode, usando l'argomento corrispondente al parametro ***mode*** per il tipo e la modalità dei permessi del nuovo file.

Il processo ***pid*** deve avere i permessi di accesso per tutte le directory che portano al file da collegare o da creare; inoltre, nell'ultima directory ci deve essere anche il permesso di scrittura, dovendo intervenire sulla stessa modificandola.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella degli inode che descrive l'inode collegato o creato. In caso di problemi, restituisce invece il puntatore nullo, aggiornando di conseguenza il contenuto della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	L'insieme degli argomenti non è valido: se l'inode è stato indicato, il parametro ***mode*** deve essere nullo; al contrario, se l'inode non è specificato, il parametro ***mode*** deve contenere informazioni valide.
`EPERM`	Non è possibile creare il collegamento di un inode corrispondente a una directory.
`EMLINK`	Non è possibile creare altri collegamenti all'inode, il quale ha già raggiunto la quantità massima.
`EEXIST`	Il file ***path*** esiste già.
`EACCES`	Impossibile accedere al percorso che dovrebbe contenere il file da collegare.
`EROFS`	Il file system è innestato in sola lettura e non si può creare il collegamento.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_inode_link.c`' [100.4.39]

VEDERE ANCHE

proc_reference(9) [99.8.7], *path_inode(9)* [99.3.36], *inode_get(9)* [99.3.23], *inode_put(9)* [99.3.24], *inode_save(9)* [99.3.26], *inode_check(9)* [99.3.16],

inode_alloc(9) [99.3.15], *inode_file_read(9)* [99.3.18], *inode_file_write(9)* [99.3.19].

99.3.38 os16: path_link(9)

«

NOME

'`path_link`' - creazione di un collegamento fisico

SINTASSI

```
<kernel/fs.h>
int path_link (pid_t pid, const char *path_old,
               const char *path_new);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`const char *`***path_old***	Il percorso del file originario.
`const char *`***path_new***	Il percorso del collegamento da creare.

DESCRIZIONE

La funzione ***path_link()*** produce un nuovo collegamento a un file già esistente. Va fornito il percorso del file già esistente, ***path_old*** e quello del file da creare, in qualità di collegamento, ***path_new***. L'operazione può avvenire soltanto se i due percorsi si trovano sulla stessa unità di memorizzazione e se ci sono i permessi di scrittura necessari nella directory di destinazione per il processo ***pid***. Dopo l'operazione di collegamento, fatta in questo modo, non è possibile distinguere quale sia stato il file originale e quale sia invece il nome aggiunto.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '`SYS_LINK`'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***path_link()*** è *link(2)* [93.23].

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di *errno*	Significato
`EINVAL`	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
`EPERM`	Operazione non consentita.
`EEXIST`	Il nome da creare esiste già.
`EACCES`	Accesso non consentito.
`ENOENT`	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
`EROFS`	Il file system consente soltanto un accesso in lettura.
`ENOTDIR`	Uno dei due percorsi non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.

FILE SORGENTI

‘`lib/unistd/link.c`’ [101.17.26]

‘`lib/sys/os16/sys.s`’ [101.12.15]

‘`kernel/proc/_isr.s`’ [100.9.1]

‘`kernel/proc/sysroutine.c`’ [100.9.30]

‘`kernel/fs/path_link.c`’ [100.4.40]

‘`kernel/fs.h`’ [100.4]

VEDERE ANCHE

link(2) [93.23], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_inode(9)* [99.3.36], *path_inode_link(9)* [99.3.37], *inode_put(9)* [99.3.24].

99.3.39 os16: path_mkdir(9)

NOME

‘**`path_mkdir`**’ - creazione di una directory

SINTASSI

```
<kernel/fs.h>
int path_mkdir (pid_t pid, const char *path, mode_t mode);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`const char *`***path***	Il percorso della directory da creare.
`mode_t` ***mode***	Modalità dei permessi da attribuire alla nuova directory.

DESCRIZIONE

La funzione ***path_mkdir()*** crea una directory, indicata attraverso un percorso (parametro ***path()***) e specificando la modalità dei permessi (parametro ***mode***). Va osservato che il valore del parametro ***mode*** non viene preso in considerazione integralmente: di questo si considerano solo gli ultimi nove bit, ovvero quelli dei permessi di utenti, gruppi e altri utenti; inoltre, vengono tolti i bit presenti nella maschera dei permessi associata al processo.

La directory che viene creata in questo modo, appartiene all'identità efficace del processo, ovvero all'utente per conto del quale questo sta funzionando.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_MKDIR`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***path_mkdir()*** è *mkdir(2)* [93.25].

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il percorso indicato non è valido.
`EEXIST`	Esiste già un file o una directory con lo stesso nome.
`ENOTDIR`	Una porzione del percorso della directory da creare, non è una directory.
`ENOENT`	Una porzione del percorso della directory da creare non esiste.
`EACCES`	Permesso negato.

FILE SORGENTI

'`lib/sys/stat/mkdir.c`' [101.13.4]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_mkdir.c`' [100.4.41]

VEDERE ANCHE

mkdir(2) [93.25], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_inode(9)* [99.3.36], *inode_file_write(9)* [99.3.19], *inode_put(9)* [99.3.24].

99.3.40 os16: path_mknod(9)

NOME

'**`path_mknod`**' - creazione di un file vuoto di qualunque tipo

SINTASSI

```
<kernel/fs.h>
int path_mknod (pid_t pid, const char *path, mode_t mode,
                dev_t device);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`const char *`***path***	Il percorso del file da creare.
`mode_t` ***mode***	Tipo e modalità dei permessi del nuovo file.
`dev_t` ***device***	Numero di dispositivo nel caso il tipo sia riferito a un file di dispositivo.

DESCRIZIONE

La funzione ***path_mknod()*** crea un file vuoto, di qualunque tipo. Potenzialmente può creare anche una directory, ma priva di qualunque voce, rendendola così non adeguata al suo scopo (una directory richiede almeno le voci '`.`' e '`..`', per potersi considerare tale).

Il parametro ***path*** specifica il percorso del file da creare; il parametro ***mode*** serve a indicare il tipo di file da creare, oltre ai permessi comuni.

Il parametro ***device***, con il quale va indicato il numero di un dispositivo (completo di numero primario e secondario), viene preso in considerazione soltanto se nel parametro ***mode*** si richiede la creazione di un file di dispositivo a caratteri o a blocchi.

Il valore del parametro ***mode*** va costruito combinando assieme delle macro-variabili definite nel file '`lib/sys/stat.h`', come descritto nella pagina di manuale *stat(2)* [93.36], tenendo conto che os16 non può gestire file FIFO, collegamenti simbolici e socket di dominio Unix.

Il valore del parametro ***mode***, per la porzione che riguarda i permessi di accesso al file, viene comunque filtrato con la maschera dei permessi (*umask(2)* [93.36]).

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_MKNOD`**'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***path_mknod()*** è *mknod(2)* [93.26].

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Il percorso indicato non è valido.
`EEXIST`	Esiste già un file o una directory con lo stesso nome.
`ENOTDIR`	Una porzione del percorso del file da creare, non è una directory.
`ENOENT`	Una porzione del percorso del file da creare non esiste.
`EACCES`	Permesso negato.

FILE SORGENTI

'`lib/sys/stat.h`' [101.13]

'`lib/sys/stat/mknod.c`' [101.13.5]

'`lib/sys/os16/sys.s`' [101.12.15]

‘kernel/proc/_isr.s’ [100.9.1]

‘kernel/proc/sysroutine.c’ [100.9.30]

‘kernel/fs.h’ [100.4]

‘kernel/fs/path_mknod.c’ [100.4.42]

VEDERE ANCHE

mknod(2) [93.26], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_inode(9)* [99.3.36], *inode_put(9)* [99.3.24].

99.3.41 os16: path_mount(9)

NOME

‘**path_mount**’, ‘**path_umount**’ - innesto e distacco di un file system

SINTASSI

```
<kernel/fs.h>
int path_mount  (pid_t pid, const char *path_dev,
                 const char *path_mnt, int options);
int path_umount (pid_t pid, const char *path_mnt);
```

ARGOMENTI

Argomento	Descrizione
pid_t ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
const char ****path_dev***	Il file di dispositivo dell’unità da innestare.
const char ****path_mnt***	Il percorso della directory di innesto.
int ***options***	Opzioni di innesto.

DESCRIZIONE

La funzione ***path_mount()*** permette l’innesto di un’unità di memorizzazione individuata attraverso il percorso del file di dispositivo nel parametro ***path_dev***, nella directory corrispondente al percorso ***path_mnt***, con le opzioni indicate numericamente nell’ultimo argomento ***options***. Le opzioni di innesto, rappresentate attraverso delle macro-variabili, sono solo due:

Opzione	Descrizione
MOUNT_DEFAULT	Innesto normale, in lettura e scrittura.
MOUNT_RO	Innesto in sola lettura.

La funzione ***path_umount()*** consente di staccare un innesto fatto precedentemente, specificando il percorso della directory in cui questo è avvenuto.

Queste funzioni vengono usate soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento delle chiamate di sistema di tipo '**SYS_MOUNT**' e '**SYS_UMOUNT**'. Le funzioni della libreria standard che si avvalgono delle chiamate di sistema che poi raggiungono ***path_mount()*** e ***path_umount()***, sono *mount(2)* [93.27] e *umount(2)* [93.27].

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Errore: va verificato il contenuto della variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
EPERM	Problema di accesso dovuto alla mancanza dei permessi necessari.
ENOTDIR	Ciò che dovrebbe essere una directory, non lo è.
EBUSY	La directory innesta già un file system e non può innestarne un altro.
ENOENT	La directory non esiste.
E_NOT_MOUNTED	La directory non innesta un file system (da staccare).
EUNKNOWN	Si è verificato un problema, non meglio precisato e non previsto.

FILE SORGENTI

'lib/sys/os16/mount.c' [101.12.12]

'lib/sys/os16/umount.c' [101.12.16]

'lib/sys/os16/sys.s' [101.12.15]

‘kernel/proc/_isr.s’ [100.9.1]

‘kernel/proc/sysroutine.c’ [100.9.30]

‘kernel/fs.h’ [100.4]

‘kernel/fs/path_mount.c’ [100.4.43]

‘kernel/fs/path_umount.c’ [100.4.45]

VEDERE ANCHE

mount(2) [93.27], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_device(9)* [99.3.33], *path_inode(9)* [99.3.36], *inode_put(9)* [99.3.24], *sb_mount(9)* [99.3.46].

99.3.42 os16: path_stat(9)

Vedere *stat(9)* [99.3.50].

99.3.43 os16: path_umount(9)

Vedere *path_mount(9)* [99.3.41].

99.3.44 os16: path_unlink(9)

NOME

‘**path_unlink**’ - cancellazione di un nome

SINTASSI

```
<kernel/fs.h>
int path_unlink (pid_t pid, const char *path);
```

ARGOMENTI

Argomento	Descrizione
pid_t ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
const char ****path***	Il percorso che rappresenta il file da cancellare.

DESCRIZIONE

La funzione ***path_unlink()*** cancella un nome da una directory, ma se si tratta dell’ultimo collegamento che ha quel file, allora libera anche l’inode corrispondente.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '`SYS_UNLINK`'. La funzione della libreria standard che si avvale della chiamata di sistema che poi raggiunge ***path_unlink()*** è *unlink(2)* [93.42].

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Errore: la variabile ***errno*** viene impostata di conseguenza.

ERRORI

Valore di ***errno***	Significato
`ENOTEMPTY`	È stata tentata la cancellazione di una directory, ma questa non è vuota.
`ENOTDIR`	Una delle directory del percorso, non è una directory.
`ENOENT`	Il nome richiesto non esiste.
`EROFS`	Il file system è in sola lettura.
`EPERM`	Mancano i permessi necessari.
`EUNKNOWN`	Si è verificato un errore imprevisto e sconosciuto.

FILE SORGENTI

'`lib/unistd/unlink.c`' [101.17.35]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/fs.h`' [100.4]

'`kernel/fs/path_unlink.c`' [100.4.46]

VEDERE ANCHE

unlink(2) [93.42], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_inode(9)* [99.3.36], *inode_check(9)* [99.3.16], *inode_file_read(9)* [99.3.18], *inode_file_write(9)* [99.3.19], *inode_put(9)* [99.3.24].

99.3.45 os16: sb_inode_status(9)

NOME

'**sb_inode_status**', '**sb_zone_status**' - verifica di utilizzazione attraverso il controllo delle mappe di inode e di zone

SINTASSI

```
<kernel/fs.h>
int sb_inode_status (sb_t *sb, ino_t ino);
int sb_zone_status  (sb_t *sb, zno_t zone);
```

ARGOMENTI

Argomento	Descrizione
sb_t *sb*	Puntatore a una voce della tabella dei super blocchi.
ino_t *ino*	Numero di inode.
zno_t *ino*	Numero di zona.

DESCRIZIONE

La funzione ***sb_inode_status()*** verifica che un certo inode, individuato per numero, risulti utilizzato nel file system a cui si riferisce il super blocco a cui punta il primo argomento.

La funzione ***sb_zone_status()*** verifica che una certa zona, individuato per numero, risulti utilizzata nel file system a cui si riferisce il super blocco a cui punta il primo argomento.

La funzione ***sb_inode_status()*** viene usata soltanto da *inode_get(9)* [99.3.23]; la funzione ***sb_zone_status()*** non viene usata affatto.

VALORE RESTITUITO

Valore	Significato
1	L'inode o la zona risultano utilizzati.
0	L'inode o la zona risultano liberi (allocabili).
−1	Errore: è stato richiesto un numero di inode o di zona pari a zero, oppure ***sb*** è un puntatore nullo.

ERRORI

Valore di *errno*	Significato
`EINVAL`	È stato richiesto un numero di inode o di zona pari a zero, oppure ***sb*** è un puntatore nullo.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/sb_inode_status.c`' [100.4.47]

'`kernel/fs/sb_zone_status.c`' [100.4.52]

VEDERE ANCHE

inode_alloc(9) [99.3.15], *zone_alloc(9)* [99.3.51].

99.3.46 os16: sb_mount(9)

«

NOME

'**`sb_mount`**' - innesto di un dispositivo di memorizzazione

SINTASSI

```
<kernel/fs.h>
sb_t *sb_mount (dev_t device, inode_t **inode_mnt,
                int options);
```

ARGOMENTI

Argomento	Descrizione
`dev_t` ***device***	Dispositivo da innestare.
`inode_t **`***inode_mnt***	Puntatore di puntatore a una voce della tabella di inode. Il valore di ****inode_mnt*** potrebbe essere un puntatore nullo.
`int` ***options***	Opzioni per l'innesto.

DESCRIZIONE

La funzione ***sb_mount()*** innesta il dispositivo rappresentato numericamente dal primo parametro, sulla directory corrispondente all'inode a cui punta, indirettamente, il secondo parametro, con le opzioni del terzo parametro.

Il secondo parametro è un puntatore di puntatore al tipo '`inode_t`', in quanto il valore rappresentato da ***inode_mnt** deve poter essere modificato dalla funzione. Infatti, quando si vuole innestare il file system principale, si crea una situazione particolare, perché la directory di innesto è la radice dello stesso file system da innestare; pertanto, ***inode_mnt** deve essere un puntatore nullo ed è compito della funzione far sì che diventi il puntatore alla voce corretta nella tabella degli inode.

Questa funzione viene usata da *proc_init(9)* [99.8.6] per innestare il file system principale, e da *path_mount(9)* [99.3.41] per innestare un file system in condizioni diverse.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella dei super blocchi che rappresenta il dispositivo innestato. In caso si insuccesso, restituisce invece il puntatore nullo e aggiorna la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EBUSY`	Il dispositivo richiesto risulta già innestato; la directory di innesto è già utilizzata; la tabella dei super blocchi è già occupata del tutto.
`EIO`	Errore di input-output.
`ENODEV`	Il file system del dispositivo richiesto non può essere gestito.
`E_MAP_INODE_TOO_BIG`	La mappa che rappresenta lo stato di utilizzo degli inode del file system, è troppo grande e non può essere caricata in memoria.
`E_MAP_ZONE_TOO_BIG`	La mappa che rappresenta lo stato di utilizzo delle zone (i blocchi di dati del file system Minix 1) è troppo grande e non può essere caricata in memoria.
`E_DATA_ZONE_TOO_BIG`	Nel file system che si vorrebbe innestare, la dimensione della zona di dati è troppo grande rispetto alle possibilità di os16.
`EUNKNOWN`	Errore imprevisto e sconosciuto.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/sb_mount.c`' [100.4.48]

VEDERE ANCHE

sb_reference(9) [99.3.47], *dev_io(9)* [99.1.1], *inode_get(9)* [99.3.23].

99.3.47 os16: sb_reference(9)

NOME

'`sb_reference`' - riferimento a un elemento della tabella dei super blocchi

SINTASSI

```
<kernel/fs.h>
sb_t *sb_reference (dev_t device);
```

ARGOMENTI

Argomento	Descrizione
`dev_t` ***device***	Dispositivo di un'unità di memorizzazione di massa.

DESCRIZIONE

La funzione ***sb_reference()*** serve a produrre il puntatore a una voce della tabella dei super blocchi. Se si fornisce il numero di un dispositivo già innestato nella tabella, si intende ottenere il puntatore alla voce relativa; se si fornisce il valore zero, si intende semplicemente avere un puntatore alla prima voce (ovvero all'inizio della tabella); se invece si fornisce il valore -1, si vuole ottenere il riferimento alla prima voce libera.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella dei super blocchi che soddisfa la richiesta. In caso di errore, restituisce invece un puntatore nullo, ma senza dare informazioni aggiuntive con la variabile ***errno***, perché il motivo è implicito nel tipo di richiesta.

ERRORI

In caso di errore la variabile ***errno*** non viene aggiornata. Tuttavia, se l'errore deriva dalla richiesta di un dispositivo di memorizzazione, significa che non è presente nella tabella; se è stato richiesta una voce libera, significa che la tabella dei super blocchi è occupata completamente.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'kernel/fs/sb_table.c' [100.4.51]

'kernel/fs/sb_reference.c' [100.4.49]

VEDERE ANCHE

inode_reference(9) [99.3.25], *file_reference(9)* [99.3.13].

99.3.48 os16: sb_save(9)

NOME

'**sb_save**' - memorizzazione di un super blocco nel proprio file system

SINTASSI

```
<kernel/fs.h>
int sb_save (sb_t *sb);
```

ARGOMENTI

Argomento	Descrizione
sb_t **sb*	Puntatore a una voce della tabella dei super blocchi in memoria.

DESCRIZIONE

La funzione ***sb_save()*** verifica se il super blocco conservato in memoria e rappresentato dal puntatore ***sb*** risulta modificato; in tal caso provvede ad aggiornarlo nell'unità di memorizzazione di origine, assieme alle mappe di utilizzo degli inode e delle zone di dati.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Errore: la variabile ***errno*** viene impostata di conseguenza.

ERRORI

Valore di ***errno***	Significato
EINVAL	Il riferimento al super blocco è un puntatore nullo.
EIO	Errore di input-output.

FILE SORGENTI

'kernel/fs.h' [100.4]

'kernel/fs/sb_save.c' [100.4.50]

VEDERE ANCHE

inode_save(9) [99.3.26], *dev_io(9)* [99.1.1].

99.3.49 os16: sb_zone_status(9)

Vedere *sb_inode_status(9)* [99.3.45].

99.3.50 os16: stat(9)

NOME

'**fd_stat**', '**path_stat**' - interrogazione dello stato di un file

SINTASSI

```
<kernel/fs.h>
int fd_stat   (pid_t pid, int fdn, struct stat *buffer);
int path_stat (pid_t pid, const char *path,
                struct stat *buffer);
```

ARGOMENTI

Argomento	Descrizione
pid_t *pid*	Il numero del processo elaborativo per conto del quale si agisce.
int *fdn*	Il numero del descrittore di file.
const char **path*	Il percorso che rappresenta il file.
struct stat **buffer*	Area di memoria in cui scrivere le informazioni sul file.

DESCRIZIONE

Le funzioni ***fd_stat()*** e ***path_stat()*** raccolgono le informazioni disponibili sul file corrispondente al descrittore ***fdn*** del processo ***pid*** o al percorso ***path***, in una variabile strutturata di tipo '**struct stat**', a cui punta ***buffer***. La struttura '**struct stat**' è definita nel file 'lib/sys/stat.h' nel modo seguente:

```
struct stat {
  dev_t      st_dev;     // Device containing the
                         // file.
  ino_t      st_ino;     // File serial number (inode
                         // number).
  mode_t     st_mode;    // File type and permissions.
  nlink_t    st_nlink;   // Links to the file.
  uid_t      st_uid;     // Owner user id.
  gid_t      st_gid;     // Owner group id.
  dev_t      st_rdev;    // Device number if it is a device
                         // file.
  off_t      st_size;    // File size.
  time_t     st_atime;   // Last access time.
  time_t     st_mtime;   // Last modification time.
  time_t     st_ctime;   // Last inode modification.
  blksize_t st_blksize; // Block size for I/O operations.
  blkcnt_t  st_blocks;  // File size / block size.
};
```

Va osservato che il file system Minix 1, usato da os16, riporta esclusivamente la data e l'ora di modifica, pertanto le altre due date previste sono sempre uguali a quella di modifica.

Il membro ***st_mode***, oltre alla modalità dei permessi che si cambiano con *fd_chmod(9)* [99.3.1], serve ad annotare anche il tipo di file. Nel file '`lib/sys/stat.h`' sono definite anche delle macro-variabili per individuare e facilitare la selezione dei bit che compongono le informazioni del membro ***st_mode***:

Modalità simbolica	Modalità numerica	Descrizione
S_IFMT	0170000_8	Maschera che raccoglie tutti i bit che individuano il tipo di file.
S_IFBLK	0060000_8	File di dispositivo a blocchi.
S_IFCHR	0020000_8	File di dispositivo a caratteri.
S_IFIFO	0010000_8	File FIFO, non gestito da os16.
S_IFREG	0100000_8	File puro e semplice.
S_IFDIR	0040000_8	Directory.
S_IFLNK	0120000_8	Collegamento simbolico, non gestito da os16.
S_IFSOCK	0140000_8	Socket di dominio Unix, non gestito da os16.

Modalità simbolica	Modalità numerica	Descrizione
S_ISUID	0004000_8	SUID.
S_ISGID	0002000_8	SGID.
S_ISVTX	0001000_8	Sticky.

Modalità simbolica	Modalità numerica	Descrizione
S_IRWXU	0000700_8	Lettura, scrittura ed esecuzione per l'utente proprietario.
S_IRUSR	0000400_8	Lettura per l'utente proprietario.
S_IWUSR	0000200_8	Scrittura per l'utente proprietario.
S_IXUSR	0000100_8	Esecuzione per l'utente proprietario.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXG`	0000070_8	Lettura, scrittura ed esecuzione per il gruppo.
`S_IRGRP`	0000040_8	Lettura per il gruppo.
`S_IWGRP`	0000020_8	Scrittura per il gruppo.
`S_IXGRP`	0000010_8	Esecuzione per il gruppo.

Modalità simbolica	Modalità numerica	Descrizione
`S_IRWXO`	0000007_8	Lettura, scrittura ed esecuzione per gli altri utenti.
`S_IROTH`	0000004_8	Lettura per gli altri utenti.
`S_IWOTH`	0000002_8	Scrittura per gli altri utenti.
`S_IXOTH`	0000001_8	Esecuzione per gli altri utenti.

os16 non considera i permessi SUID (*Set user id*), SGID (*Set group id*) e Sticky; inoltre, non considera i permessi legati al gruppo, perché non tiene traccia dei gruppi.

Queste funzioni vengono usate soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento delle chiamate di sistema di tipo '**`SYS_STAT`**' e '**`SYS_FSTAT`**'. Le funzioni della libreria standard che si avvalgono delle chiamate di sistema che poi raggiungono ***fd_stat()*** e ***path_stat()***, sono *fstat(2)* [93.36] e *stat(2)* [93.36].

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
−1	Errore, descritto dal contenuto della variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
`ENFILE`	Troppi file aperti nel sistema.
`ENOENT`	File non trovato.
`EACCES`	Permesso negato.
`EBADF`	Il descrittore del file richiesto non è valido.

FILE SORGENTI

‘`lib/sys/stat/fstat.c`’ [101.13.3]

‘`lib/sys/stat/stat.c`’ [101.13.6]

‘`kernel/proc.h`’ [100.9]

‘`kernel/proc/_isr.s`’ [100.9.1]

‘`kernel/proc/sysroutine.c`’ [100.9.30]

‘`kernel/fs.h`’ [100.4]

‘`kernel/fs/fd_stat.c`’ [100.4.11]

‘`kernel/fs/path_stat.c`’ [100.4.44]

VEDERE ANCHE

fstat(2) [93.36], *stat(2)* [93.36], *sysroutine(9)* [99.8.28], *proc_reference(9)* [99.8.7], *path_inode(9)* [99.3.36], *inode_put(9)* [99.3.24].

99.3.51 os16: zone_alloc(9)

«

NOME

‘**`zone_alloc`**’, ‘**`zone_free`**’ - allocazione di zone di dati

SINTASSI

```
<kernel/fs.h>
zno_t zone_alloc (sb_t *sb);
int   zone_free  (sb_t *sb, zno_t zone);
```

ARGOMENTI

Argomento	Descrizione
sb_t ****sb***	Puntatore a una voce della tabella dei super blocchi in memoria.
zno_t ***zone***	Numero di zona da liberare.

DESCRIZIONE

La funzione ***zone_alloc()*** occupa una zona nella mappa associata al super blocco a cui si riferisce ***sb***, restituendone il numero. La funzione ***zone_free()*** libera una zona che precedentemente risultava occupata nella mappa relativa.

VALORE RESTITUITO

La funzione ***zone_alloc()*** restituisce il numero della zona allocata. Se questo numero è zero, si tratta di un errore, e va considerato il contenuto della variabile ***errno***.

La funzione ***zone_free()*** restituisce zero in caso di successo, oppure −1 in caso di errore, aggiornando di conseguenza la variabile ***errno***.

ERRORI

Valore di ***errno***	Significato
EROFS	Il file system è innestato in sola lettura, pertanto non è possibile apportare cambiamenti alla mappa di utilizzo delle zone.
ENOSPC	Non è possibile allocare una zona, perché non ce ne sono di libere.
EINVAL	L'argomento corrispondente a ***sb*** è un puntatore nullo; la zona di cui si richiede la liberazione è precedente alla prima zona dei dati (pertanto non può essere liberata, in quanto riguarda i dati amministrativi del super blocco); la zona da liberare è successiva allo spazio gestito dal file system.
EUNKNOWN	Errore imprevisto e sconosciuto.

FILE SORGENTI

'kernel/fs.h' [100.4]

'kernel/fs/zone_alloc.c' [100.4.53]

'kernel/fs/zone_free.c' [100.4.54]

VEDERE ANCHE

zone_write(9) [99.3.53], *sb_save(9)* [99.3.48].

99.3.52 os16: zone_free(9)

Vedere *zone_alloc(9)* [99.3.51].

99.3.53 os16: zone_read(9)

NOME

'**zone_read**', '**zone_write**' - lettura o scrittura di una zona di dati

SINTASSI

```
<kernel/fs.h>
int zone_read  (sb_t *sb, zno_t zone, void *buffer);
int zone_write (sb_t *sb, zno_t zone, void *buffer);
```

ARGOMENTI

Argomento	Descrizione
sb_t *sb*	Puntatore a una voce della tabella dei super blocchi in memoria.
zno_t *zone*	Numero di zona da leggere o da scrivere
void *buffer*	Puntatore alla posizione iniziale in memoria dove depositare la zona letta o da dove trarre i dati per la scrittura della zona.

DESCRIZIONE

La funzione ***zone_read()*** legge una zona e ne trascrive il contenuto a partire da ***buffer***. La funzione ***zone_write()*** scrive una zona copiandovi al suo interno quanto si trova in memoria a partire da ***buffer***. La zona è individuata dal numero ***zone*** e riguarda il file system a cui si riferisce il super blocco ***sb***.

La lettura o la scrittura riguarda una zona soltanto, ma nella sua interezza.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	Gli argomenti non sono validi.
`EROFS`	Il file system è innestato in sola lettura.
`EIO`	Errore di input-output.

FILE SORGENTI

'`kernel/fs.h`' [100.4]

'`kernel/fs/zone_read.c`' [100.4.55]

'`kernel/fs/zone_write.c`' [100.4.56]

VEDERE ANCHE

zone_alloc(9) [99.3.51], *zone_free(9)* [99.3.51].

99.4 os16: ibm_i86(9)

«

Il file '`kernel/ibm_i86.h`' [100.5] descrive le funzioni e le macro-istruzioni per la gestione dell'hardware.

La sezione 90.3 descrive complessivamente queste funzioni e le tabelle successive sono tratte da lì.

Tabella 90.26. Funzioni e macro-istruzioni di basso livello, dichiarate nel file di intestazione '`kernel/ibm_i86.h`' e descritte nei file della directory '`kernel/ibm_i860/`'. Le macro-istruzioni hanno argomenti di tipo numerico non precisato, purché in grado di rappresentare il valore necessario.

Funzione o macro-istruzione	Descrizione

`void _int10_00 (uint16_t` ***video_mode***`);`
 	
`void int10_00 (`***video_mode***`);`
	Imposta la modalità video della console.
	Questa funzione viene usata solo da ***con_init()***, per inizializzare la console; la modalità video è stabilita dalla macro-variabile ***IBM_I86_VIDEO_MODE***, dichiarata nel file '`kernel/ibm_i86.h`'.

Funzione o macro-istruzione	Descrizione
`void _int10_02 (uint16_t page, uint16_t position);` `void int10_02 (page, position);`	Colloca il cursore in una posizione determinata dello schermo, relativo a una certa pagina video. Questa funzione viene usata solo da ***con_putc()***.
`void _int10_05 (uint16_t page);` `void int10_05 (page);`	Seleziona la pagina attiva del video. Questa funzione viene usata solo da ***con_init()*** e ***con_select()***.
`void _int12 (void);` `void int12 (void);`	Restituisce la quantità di memoria disponibile, in multipli di 1024 byte.
`void _int13_00 (uint16_t drive);` `void int13_00 (drive);`	Azzera lo stato dell'unità a disco indicata, rappresentata da un numero secondo le convenzioni del BIOS. Viene usata solo dalle funzioni '`dsk_...()`' che si occupano dell'accesso alle unità a disco.
`uint16_t _int13_02 (uint16_t drive, uint16_t sectors, uint16_t cylinder, uint16_t head, uint16_t sector, void *buffer);` `void int13_02 (drive, sectors, cylinder, head, sector, buffer);`	Legge dei settori da un'unità a disco. Questa funzione viene usata soltanto da ***dsk_read_sectors()***.

Funzione o macro-istruzione	Descrizione

`uint16_t _int13_03 (uint16_t` ***drive***`,`
`uint16_t` ***sectors***`,`
`uint16_t` ***cylinder***`,`
`uint16_t` ***head***`,`
`uint16_t` ***sector***`,`
`void *`***buffer***`);`
`void int13_03 (`***drive***`,` ***sectors***`,`
cylinder`,` ***head***`,`
sector`,` ***buffer***`);`
	Scrive dei settori in un'unità a disco.
	Questa funzione viene usata solo da ***dsk_write_sectors()***.

`uint16_t _int16_00 (void);` `void int16_00 (void);`	Legge un carattere dalla tastiera, rimuovendolo dalla memoria tampone relativa. Viene usata solo in alcune funzioni di controllo della console, denominate '**`con_...()`**'.
`uint16_t _int16_01 (void);` `void int16_01 (void);`	Verifica se è disponibile un carattere dalla tastiera: se c'è ne restituisce il valore, ma senza rimuoverlo dalla memoria tampone relativa, altrimenti restituisce zero. Viene usata solo dalle funzioni di gestione della console, denominate '**`con_...()`**'.
`void _int16_02 (void);` `void int16_02 (void);`	Restituisce un valore con cui è possibile determinare quali funzioni speciali della tastiera risultano inserite (inserimento, fissa-maiuscole, blocco numerico, ecc.). Al momento la funzione non viene usata.
`uint16_t _in_8 (uint16_t` ***port***`);` `void in_8 (`***port***`);`	Legge un byte dalla porta di I/O indicata. Questa funzione viene usata da ***irq_on()***, ***irq_off()*** e ***dev_mem()***.
`uint16_t _in_16 (uint16_t` ***port***`);` `void in_16 (`***port***`);`	Legge un valore a 16 bit dalla porta di I/O indicata. Questa funzione viene usata solo da ***dev_mem()***.

Funzione o macro-istruzione	Descrizione

`void _out_8 (uint16_t` ***port***`,`
`uint16_t` ***value***`);`
`void out_8 (`***port***`,` ***value***`);`
	Scrive un byte nella porta di I/O indicata. Questa funzione viene usata da ***irq_on()***, ***irq_off()*** e ***dev_mem()***.

`void _out_16 (uint16_t` ***port***`,`
`uint16_t` ***value***`);`
`void out_16 (`***port***`,` ***value***`);`
	Scrive un valore a 16 bit nella porta indicata. Questa funzione viene usata solo da ***dev_mem()***.

`void cli (void);`	Azzera l'indicatore delle interruzioni, nel registro ***FLAGS***. La funzione serve a permettere l'uso dell'istruzione '**`CLI`**' dal codice in linguaggio C, ma in questa veste, viene usata solo dalla funzione ***proc_init()***.
`void sti (void);`	Attiva l'indicatore delle interruzioni, nel registro ***FLAGS***. La funzione serve a permettere l'uso dell'istruzione '**`STI`**' dal codice in linguaggio C, ma in questa veste, viene usata solo dalla funzione ***proc_init()***.
`void irq_on (unsigned int` ***irq***`);`	Abilita l'interruzione hardware indicata. Questa funzione viene usata solo da ***proc_init()***.
`void irq_off (unsigned int` ***irq***`);`	Disabilita l'interruzione hardware indicata. Questa funzione viene usata solo da ***proc_init()***.

Funzione o macro-istruzione	Descrizione
`void _ram_copy (segment_t` ***org_seg*** `,` `offset_t` ***org_off*** `,` `segment_t` ***dst_seg*** `,` `offset_t` ***dst_off*** `,` `uint16_t` ***size*** `);` `void ram_copy (`***org_seg*** `,` ***org_off*** `,` ***dst_seg*** `,` ***dst_off*** `,` ***size*** `);`	Copia una certa quantità di byte, da una posizione di memoria all'altra, specificando segmento e scostamento di origine e destinazione. Viene usata solo dalle funzioni '**mem_...()**'.

Tabella 90.27. Funzioni per l'accesso alla console, dichiarate nel file di intestazione '`kernel/ibm_i86.h`' e descritte nei file contenuti nella directory '`kernel/ibm_i86/`'.

Funzione	Descrizione
`int con_char_read (void);`	Legge un carattere dalla console, se questo è disponibile, altrimenti restituisce il valore zero. Questa funzione viene usata solo da ***proc_sch_terminals()***.
`int con_char_wait (void);`	Legge un carattere dalla console, ma se questo non è ancora disponibile, rimane in attesa, bloccando tutto il sistema operativo. Questa funzione non è utilizzata.
`int con_char_ready (void);`	Verifica se è disponibile un carattere dalla console: se è così, restituisce un valore diverso da zero, corrispondente al carattere in attesa di essere prelevato. Questa funzione viene usata solo da ***proc_sch_terminals()***.
`void con_init (void);`	Inizializza la gestione della console. Questa funzione viene usata solo da ***tty_init()***.
`void con_select (int` ***console*** `);`	Seleziona la console desiderata, dove la prima si individua con lo zero. Questa funzione viene usata solo da ***tty_console()***.

Funzione	Descrizione
void con_putc (int ***console***, int ***c***);	Visualizza il carattere indicato sullo schermo della console specificata, sulla posizione in cui si trova il cursore, facendolo avanzare di conseguenza e facendo scorrere il testo in alto, se necessario. Questa funzione viene usata solo da ***tty_write()***.
void con_scroll (int ***console***);	Fa avanzare in alto il testo della console selezionata. Viene usata internamente, solo dalla funzione ***con_putc()***.

Tabella 90.30. Funzioni per l'accesso ai dischi, dichiarate nel file di intestazione 'kernel/ibm_i86.h'.

Funzione	Descrizione
void dsk_setup (void);	Predispone il contenuto dell'array ***dsk_table[]***. Questa funzione viene usata soltanto da ***main()***.
int dsk_reset (int ***drive***);	Azzera lo stato dell'unità corrispondente a ***dsk_table[drive].bios_drive***. Viene usata solo internamente, dalle altre funzioni '**dsk_...()**'.
void dsk_sector_to_chs (int ***drive***, unsigned int ***sector***, dsk_chs_t ****chs***);	Modifica le coordinate della variabile strutturata a cui punta l'ultimo parametro, con le coordinate corrispondenti al numero di settore fornito. Viene usata solo internamente, dalle altre funzioni '**dsk_...()**'.
int dsk_read_sectors (int ***drive***, unsigned int ***start_sector***, void ****buffer***, unsigned int ***n_sectors***);	Legge una sequenza di settori da un disco, mettendo i dati in memoria, a partire dalla posizione espressa da un puntatore generico. La funzione è ricorsiva, ma oltre che da se stessa, viene usata internamente da ***dsk_read_bytes()*** e da ***dsk_write_bytes()***.

Funzione	Descrizione
`int dsk_write_sectors (int` ***drive***`, unsigned int` ***start_sector***`, void *`***buffer***`, unsigned int` ***n_sectors***`);`	Scrive una sequenza di settori in un disco, traendo i dati da un puntatore a una certa posizione della memoria. La funzione è ricorsiva, ma oltre che da se stessa, viene usata solo internamente da ***dsk_write_bytes()***.
`size_t dsk_read_bytes (int` ***drive***`, off_t` ***offset***`, void *`***buffer***`, size_t count);`	Legge da una certa unità a disco una quantità specificata di byte, a partire dallo scostamento indicato (nel disco), il quale deve essere un valore positivo. Questa funzione viene usata solo da ***dev_dsk()***.
`size_t dsk_write_bytes (int` ***drive***`, off_t` ***offset***`, void *`***buffer***`, size_t count);`	Scrive su una certa unità a disco una quantità specificata di byte, a partire dallo scostamento indicato (nel disco), il quale deve essere un valore positivo. Questa funzione viene usata solo da ***dev_dsk()***.

99.5 os16: k_libc(9)

«

Il file '`kernel/k_libc.h`' [100.6] descrive alcune funzioni con nomi che iniziano per '**`k_`**...' (dove la lettera «k» sta per kernel) e riproducono il comportamento di funzioni standard, della libreria C. Per esempio, ***k_printf()*** è l'equivalente di ***printf()***, ma per la gestione interna del kernel.

Teoricamente, quando una funzione interna al kernel può ricondursi allo standard, dovrebbe avere il nome previsto. Tuttavia, per evitare di dover qualificare ogni volta l'ambito di una funzione, sono stati usati nomi differenti, ciò anche al fine di non creare complicazioni in fase di compilazione di tutto il sistema.

99.6 os16: main(9)

«

Il file '`kernel/main.h`' [100.7] descrive la funzione ***main()*** del kernel e altre funzioni accessorie, assieme al codice iniziale necessario per mettere in funzione il kernel stesso.

Si rimanda alla sezione 90.2 che descrive dettagliatamente il codice iniziale del kernel.

99.7 os16: memory(9)

«

Il file '`kernel/memory.h`' [100.8] descrive le funzioni per la gestione della memoria, a livello di sistema.

Per la descrizione dell'organizzazione della gestione della memoria si rimanda alla sezione 90.4. Le tabelle successive che sintetizzano l'uso delle funzioni di questo gruppo, sono tratte da quel capitolo.

Tabella 90.33. Funzioni per la gestione della mappa della memoria, dichiarate nel file di intestazione '`kernel/memory.h`' e realizzate nella directory '`kernel/memory/`'.

Funzione	Descrizione
`uint16_t *mb_reference (void);`	Restituisce il puntatore alla tabella dei blocchi di memoria, per uniformare l'accesso alla tabella dalle funzioni che non fanno parte del gruppo contenuto nella directory '`kernel/memory/`'.
`ssize_t mb_alloc (addr_t` ***address***`,` `size_t` ***size***`);`	Alloca la memoria a partire dall'indirizzo efficace indicato, per la quantità di byte richiesta (zero corrisponde a 10000_{16} byte). L'allocazione ha termine anticipatamente se si incontra un blocco già utilizzato. La funzione restituisce la dimensione allocata effettivamente.

Funzione	Descrizione
`ssize_t mb_free (addr_t `***address***`,` `size_t `***size***`);`	Libera la memoria a partire dall'indirizzo efficace indicato, per la quantità di byte richiesta (zero corrisponde a 10000_{16} byte). Lo spazio viene liberato in ogni caso, anche se risulta già libero; tuttavia viene prodotto un avvertimento a video se si verifica tale ipotesi.

`int mb_alloc_size`
`(size_t `***size***`,`
`memory_t *`***allocated***`);`
	Cerca e alloca un'area di memoria della dimensione richiesta, modificando la variabile strutturata di cui viene fornito il puntatore come secondo parametro. In pratica, l'indirizzo e l'estensione della memoria allocata effettivamente si trovano nella variabile strutturata in questione, mentre la funzione restituisce zero (se va tutto bene) o −1 se non è disponibile la memoria libera richiesta.

Tabella 90.34. Funzioni per le operazioni di lettura e scrittura in memoria, dichiarate nel file di intestazione '`kernel/memory.h`' e realizzate nella directory '`kernel/memory/`'.

Funzione	Descrizione

`void mem_copy (addr_t `***orig***`,`
`addr_t `***dest***`,`
`size_t `***size***`);`
	Copia la quantità richiesta di byte, dall'indirizzo di origine a quello di destinazione, espressi in modo efficace.

`size_t mem_read (addr_t `***start***`,`
`void *`***buffer***`,`
`size_t `***size***`);`
	Legge dalla memoria, a partire dall'indirizzo indicato come primo parametro, la quantità di byte indicata come ultimo parametro. Ciò che viene letto va poi copiato nella memoria tampone corrispondente al puntatore generico indicato come secondo parametro.

Funzione	Descrizione
`size_t mem_write (addr_t` ***start*** `, void *`***buffer*** `, size_t` ***size*** `);`	Scrive, in memoria, a partire dall'indirizzo indicato come primo parametro, la quantità di byte indicata come ultimo parametro. Ciò che viene scritto proviene dalla memoria tampone corrispondente al puntatore generico indicato come secondo parametro.

99.8 os16: proc(9)

«

Il file '`kernel/proc.h`' [100.9] descrive ciò che serve per la gestione dei processi. In modo particolare, in questo file si definisce il tipo derivato '**`proc_t`**', con cui si realizza la tabella dei processi.

Figura 90.80. Struttura del tipo '**`proc_t`**', corrispondente agli elementi dell'array ***proc_table[]***.

ppid: pid genitore
pgrp: pid del gruppo
uid: identità reale
euid: identità efficace
sid: identitàsalvata
terminale di controllo
cwd: directory corrente in forma di percorso
cwd: directory corrente in forma di puntatore a inode
umask
segnali attivati
segnali inibiti
tempo di utilizzo in quantità di impulsi
stato del processo
eventi attesi
segnale atteso
tempo da attendere
indirizzo efficace delle istruzioni
segmento istruzioni
dimensione istruzioni
indirizzo efficace dei dati
segmento dati
dimensione dati
sp: indice della pila
valore restituito al termine
nome del processo
tabella dei descrittori dei file del processo

Listato 90.81. Struttura del tipo '**`proc_t`**', corrispondente agli elementi dell'array ***proc_table[]***.

```
typedef struct {
  pid_t               ppid;
  pid_t               pgrp;
  uid_t               uid;
  uid_t               euid;
  uid_t               suid;
  dev_t               device_tty;
  char                path_cwd[PATH_MAX];
  inode_t            *inode_cwd;
  int                 umask;
  unsigned long int sig_status;
  unsigned long int sig_ignore;
  clock_t             usage;
  unsigned int        status;
  int                 wakeup_events;
  int                 wakeup_signal;
  unsigned int        wakeup_timer;
  addr_t              address_i;
  segment_t           segment_i;
  size_t              size_i;
  addr_t              address_d;
  segment_t           segment_d;
  size_t              size_d;
  uint16_t            sp;
  int                 ret;
  char                name[PATH_MAX];
  fd_t                fd[FOPEN_MAX];
} proc_t;
```

Tabella 90.82. Membri del tipo '**`proc_t`**'.

Membro	Contenuto
`ppid`	Numero del processo genitore: *parent process id*.
`pgrp`	Numero del gruppo di processi a cui appartiene quello della voce corrispondente: *process group*. Si tratta del numero del processo a partire dal quale viene definito il gruppo.

Membro	Contenuto
`uid`	Identità reale del processo della voce corrispondente: *user id*. Si tratta del numero dell'utente, secondo la classificazione del file '`/etc/passwd`', per conto del quale il processo è stato avviato. Tuttavia, i privilegi del processo dipendono dall'identità efficace, definita dal membro '**`euid`**'.
`euid`	Identità efficace del processo della voce corrispondente: *effective user id*. Si tratta del numero dell'utente, secondo la classificazione del file '`/etc/passwd`', per conto del quale il processo è in funzione; pertanto, il processo ha i privilegi di quell'utente.
`suid`	Identità salvata: *saved user id*. Si tratta del valore che aveva ***euid*** prima di cambiare identità.
`device_tty`	Terminale di controllo, espresso attraverso il numero del dispositivo.
`path_cwd` `inode_cwd`	Entrambi i membri rappresentano la directory corrente del processo: nel primo caso in forma di percorso, ovvero di stringa, nel secondo in forma di puntatore a inode rappresentato in memoria.
`umask`	Maschera dei permessi associata al processo: i permessi attivi nella maschera vengono tolti in fase di creazione di un file o di una directory.
`sig_status`	Segnali inviati al processo e non ancora trattati: ogni segnale si associa a un bit differente del valore del membro ***sig_status***; un bit a uno indica che il segnale corrispondente è stato ricevuto e non ancora trattato.
`sig_ignore`	Segnali che il processo ignora: ogni segnale da ignorare si associa a un bit differente del valore del membro ***sig_ignore***; un bit a uno indica che quel segnale va ignorato.
`usage`	Tempo di utilizzo della CPU, da parte del processo, espresso in impulsi del temporizzatore, il quale li produce alla frequenza di circa 18,2 Hz.

Membro	Contenuto
`status`	Stato del processo, rappresentabile attraverso una macro-variabile simbolica, definita nel file '`proc.h`'. Per os16, gli stati possibili sono: «inesistente», quando si tratta di una voce libera della tabella dei processi; «creato», quando un processo è appena stato creato; «pronto», quando un processo è pronto per essere eseguito, «in esecuzione», quando il processo è in funzione; «sleeping», quando un processo è in attesa di qualche evento; «zombie», quando un processo si è concluso, ha liberato la memoria, ma rimangono le sue tracce perché il genitore non ha ancora recepito la sua fine.
`wakeup_events`	Eventi attesi per il risveglio del processo, ammesso che si trovi nello stato si attesa. Ogni tipo di evento che può essere atteso corrisponde a un bit e si rappresenta con una macro-variabile simbolica, dichiarata nel file '`lib/sys/os16.h`'.
`wakeup_signal`	Ammesso che il processo sia in attesa di un segnale, questo membro esprime il numero del segnale atteso.
`wakeup_timer`	Ammesso che il processo sia in attesa dello scadere di un conto alla rovescia, questo membro esprime il numero di secondi che devono ancora trascorrere.

`address_i`
`segment_i`
`size_i`
	Il valore di questi membri descrive la memoria utilizzata dal processo per le istruzioni (il segmento codice). Le informazioni sono in parte ridondanti, perché conoscendo ***segment_i*** si ottiene facilmente ***address_i*** e viceversa, ma ciò consente di ridurre i calcoli nelle funzioni che ne fanno uso.

`address_d`
`segment_d`
`size_d`
	Il valore di questi membri descrive la memoria utilizzata dal processo per i dati (il segmento usato per le variabili statiche e per la pila). Anche in questo caso, le informazioni sono in parte ridondanti, ma ciò consente di semplificare il codice nelle funzioni che ne fanno uso.

`sp`	Indice della pila dei dati, nell'ambito del segmento dati del processo. Il valore è significativo quando il processo è nello stato di pronto o di attesa di un evento. Quando invece un processo era attivo e viene interrotto, questo valore viene aggiornato.

Membro	Contenuto
`ret`	Rappresenta il valore restituito da un processo terminato e passato nello stato di «zombie».
`name`	Il nome del processo, rappresentato dal nome del programma avviato.
`fd`	Tabella dei descrittori dei file relativi al processo.

99.8.1 os16: isr_1C(9)

NOME

'**`isr_1C`**', '**`isr_80`**' - routine di gestione delle interruzioni

DESCRIZIONE

La routine '**`isr_1C`**' del file '`kernel/proc/_isr.s`' viene eseguita a ogni impulso del temporizzatore, proveniente dal sistema delle interruzioni hardware; la routine '**`isr_80`**', in modo analogo, viene eseguita in corrispondenza dell'interruzione software 80_{16} Perché ciò avvenga, nella tabella IVT, nelle voci che riguardano l'interruzione $1C_{16}$ e 80_{16}, si trova l'indirizzo corrispondente alle routine in questione. La configurazione della tabella IVT avviene per mezzo della funzione *ivt_load(9)* [99.8.2].

La routine '**`isr_1C`**' prevede il salvataggio dei registri principali nella pila dei dati in funzione al momento dell'interruzione. Quindi vengono modificati i registri che definiscono l'area dati (ES e DS) e successivamente ciò permette di intervenire su delle variabili locali: viene incrementato il contatore degli impulsi del temporizzatore; viene incrementato il contatore dei secondi, se il contatore degli impulsi è divisibile per 18 senza dare resto; vengono salvati l'indice e il segmento della pila dei dati, in due variabili locali.

Dalla verifica del valore del segmento in cui si colloca la pila dei dati del processo interrotto, la routine verifica se si tratta di un processo comune o del kernel. Se si tratta di un processo comune, si scambia la pila con quella del kernel. Per questo la routine si avvale della variabile ***_ksp*** (*kernel stack pointer*), usata anche dalla funzione *proc_scheduler(9)* [99.8.11]. Sempre se si tratta dell'interruzione di un processo diverso dal kernel, viene chiamata la funzione ***proc_scheduler()***, già citata, fornendo come argomenti il puntatore alla variabile che contiene l'indice della pila e il puntatore alla variabile che contiene il segmento di memoria che ospita la pila dei dati. Al termine viene scambiata nuovamente la pila dei dati, usando come valori quanto contenuto nelle variabili che prima sono servite per salvare l'indice e il segmento della pila.

Poi, indipendentemente dal tipo di processo, vengono ripristinati i registri accumulati in precedenza nella pila e viene restituito il controllo, concludendo il lavoro dell'interruzione.

Va osservato che la funzione ***proc_scheduler()*** riceve l'indice e il segmento della pila dei dati attraverso dei puntatori a variabili scalari. Pertanto, tale funzione è perfettamente in grado di sostituire questi valori, con quelli della pila di un altro processo. Per questo, quando al ritorno della funzione viene ripristinata la pila sulla base di tali variabili, si ha uno scambio di processi. Il ripristino successivo dalla pila dei registri, completa il procedimento di sostituzione dei processi.

La routine '**`isr_80`**' viene attivata da un'interruzione software, dovuta a una chiamata di sistema. Questa routine si distingue leggermente da '**`isr_1C`**', in quanto non si occupa di tenere conto del tempo trascorso, ma ha la necessità di recuperare dalla pila del processo interrotto, i valori che hanno origine dalla chiamata di sistema. Si tratta sempre del numero della chiamata di sistema, del puntatore al messaggio trasmesso con la chiamata e della sua lunghezza.

Si può osservare anche un'altra differenza importante, per cui, se l'interruzione riguarda il processo del kernel, l'indice della pila dello stesso viene conservato nella variabile ***_ksp***. Questo fatto è importante, perché prima di abilitare la gestione delle interruzioni, è necessario che il kernel stesso ne provochi una, in modo da poter salvare la prima volta l'indice della propria pila.

Successivamente, indipendentemente dal processo interrotto, si chiama la funzione *sysroutine(9)* [99.8.28], alla quale si passano come argomenti, oltre che i puntatori all'indice e al segmento della pila dei dati del processo interrotto, anche gli argomenti della chiamata di sistema.

La funzione ***sysroutine()*** si avvale a sua volta della funzione ***proc_scheduler()***, pertanto anche in questo caso la pila dei dati che viene ripristinata successivamente può risultare differente da quella del processo interrotto originariamente, comportando anche in questo caso lo scambio del processo con un altro.

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_table.c`' [100.9.29]

'`kernel/proc/_isr.s`' [100.9.1]

VEDERE ANCHE

ivt_load(9) [99.8.2], *sys(2)* [93.37], *proc_scheduler(9)* [99.8.11], *sysroutine(9)* [99.8.28].

99.8.2 os16: ivt_load(9)

NOME

'**ivt_load**' - caricamento della tabella IVT

SINTASSI

```
<kernel/proc.h>
void _ivt_load (void);
```

```
<kernel/proc.h>
void  ivt_load (void);
```

DESCRIZIONE

La funzione ***_ivt_load()***, ovvero la macro-istruzione corrispondente ***ivt_load()***, modifica la tabella IVT del BIOS, in modo che nella posizione corrispondente all'interruzione $1C_{16}$ ci sia il puntatore alla routine *isr_1C(9)* [99.8.1], e che in corrispondenza dell'interruzione 80_{16} ci sia il puntatore alla routine *isr_80(9)* [99.8.1].

Questa funzione viene usata una volta sola, all'interno di *main(9)* [99.6].

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/_ivt_load.s`' [100.9.2]

VEDERE ANCHE

sys(2) [93.37], *isr_80(9)* [99.8.1], *proc_scheduler(9)* [99.8.11], *sysroutine(9)* [99.8.28].

99.8.3 os16: proc_available(9)

NOME

'**proc_available**' - inizializzazione di un processo libero

SINTASSI

```
<kernel/proc.h>
void proc_available (pid_t pid);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo da inizializzare.

DESCRIZIONE

La funzione ***proc_available()*** si limita a inizializzare, con valori appropriati, i dati di un processo nella tabella relativa, in modo che risulti correttamente uno spazio libero per le allocazioni successive.

Questa funzione viene usata da *proc_init(9)* [99.8.6], *proc_sig_chld(9)* [99.8.12], *proc_sys_wait(9)* [99.8.27].

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_table.c`' [100.9.29]

'`kernel/proc/proc_available.c`' [100.9.3]

99.8.4 os16: proc_dump_memory(9)

«

NOME

'**`proc_dump_memory`**' - copia di una porzione di memoria in un file

SINTASSI

```
<kernel/proc.h>
void proc_dump_memory (pid_t pid, addr_t address, size_t size,
                       char *name);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo elaborativo per conto del quale si agisce.
`addr_t` ***address***	Indirizzo efficace della memoria.
`size_t` ***size***	Quantità di byte da trascrivere, a partire dall'indirizzo efficace.
`char *`***name***	Nome del file da creare.

DESCRIZIONE

La funzione ***proc_dump_memory()*** salva in un file una porzione di memoria, secondo le coordinate fornita dagli argomenti.

Viene usata esclusivamente da *proc_sig_core(9)* [99.8.6], quando si riceve un segnale per cui è necessario scaricare la memoria di un processo. In quel caso, se il processo eliminato ha i permessi per scrivere nella directory radice, vengono creati due file: uno con l'immagine del segmento codice ('`/core.i`') e l'altro con l'immagine del segmento dati ('`/core.d`').

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_sig_core.c`' [100.9.14]

VEDERE ANCHE

fd_open(9) [99.3.8], *fd_write(9)* [99.3.12], *fd_close(9)* [99.3.3].

99.8.5 os16: proc_find(9)

«

NOME

'**`proc_find`**' - localizzazione di un processo sulla base dell'indirizzo del segmento dati

SINTASSI

```
<kernel/proc.h>
pid_t proc_find (segment_t segment_d);
```

ARGOMENTI

Argomento	Descrizione
`segment_t` ***segment_d***	Indirizzo del segmento da cercare nella tabella dei processi, come allocato per il segmento dati.

DESCRIZIONE

La funzione ***proc_find()*** scandisce la tabella dei processi, alla ricerca di quel processo il cui segmento dati corrisponde al valore fornito come argomento. Ciò serve per sapere chi sia il processo interrotto, del quale si conosce il valore che, prima dell'interruzione, aveva il registro DS (*data segment*).

Questa funzione viene usata da *proc_scheduler(9)* [99.8.11] e da *sysroutine(9)* [99.8.28].

VALORE RESTITUITO

La funzione restituisce il numero del processo trovato e non è ammissibile che la ricerca possa fallire. Infatti, se così fosse, si produrrebbe un errore fatale, con avvertimento a video, tale da arrestare il funzionamento del kernel.

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_table.c`' [100.9.29]

'`kernel/proc/proc_find.c`' [100.9.5]

99.8.6 os16: proc_init(9)

«

NOME

'**`proc_init`**' - inizializzazione della gestione complessiva dei processi elaborativi

SINTASSI

```
<kernel/proc.h>
extern uint16_t _etext;
void proc_init (void);
```

ARGOMENTI

Argomento	Descrizione
`extern uint16_t` ***_etext;***	La variabile ***_etext*** viene fornita dal compilatore e rappresenta l'indirizzo in cui l'area codice si è conclusa. Il valore di ***_etext*** si riferisce a un'area codice che inizia dall'indirizzo zero, pertanto questo dato viene usato per conoscere la dimensione dell'area codice del kernel.

DESCRIZIONE

La funzione ***proc_init()*** viene usata una volta sola, dalla funzione *main(9)* [99.6], per predisporre la gestione dei processi. Per la precisione svolge le operazioni seguenti:

- carica la tabella IVT, in modo che le interruzioni software $1C_{16}$ e 80_{16} siano dirette correttamente al codice che deve gestirle;
- programma il temporizzatore interno, in modo da produrre una frequenza di circa 18,2 Hz;

- inizializza la tabella dei processi in modo che tutti gli alloggiamenti previsti risultino liberi;
- innesta il file system principale, presupponendo che possa trattarsi soltanto della prima unità a dischetti;
- inizializza correttamente le voci del processo zero, ovvero quelle del kernel, segnando anche come allocata la porzione di memoria utilizzata dal kernel e lo spazio iniziale usato dal BIOS (tabella IVT e BDA);
- abilita le interruzioni hardware del temporizzatore, della tastiera e dell'unità a dischetti: le altre interruzioni hardware rimangono disabilitate.

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_table.c`' [100.9.29]

'`kernel/proc/proc_init.c`' [100.9.6]

VEDERE ANCHE

ivt_load(9) [99.8.2], *proc_available(9)* [99.8.3], *sb_mount(9)* [99.3.46].

99.8.7 os16: proc_reference(9)

«

NOME

'**`proc_reference`**' - puntatore alla voce che rappresenta un certo processo

SINTASSI

```
<kernel/proc.h>
proc_t *proc_reference (pid_t pid);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il numero del processo cercato nella tabella relativa.

DESCRIZIONE

La funzione ***proc_reference()*** serve a produrre il puntatore all'elemento dell'array ***proc_table[]*** che contiene i dati del processo indicato per numero come argomento.

Viene usata dalle funzioni che non fanno parte del gruppo di '`kernel/proc.h`'.

VALORE RESTITUITO

Restituisce il puntatore all'elemento della tabella ***proc_table[]*** che rappresenta il processo richiesto. Se il numero del processo richiesto non può esistere, la funzione restituisce il puntatore nullo '**`NULL`**'.

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_table.c`' [100.9.29]

'`kernel/proc/proc_reference.c`' [100.9.7]

99.8.8 os16: proc_sch_signals(9)

«

NOME

'**`proc_sch_signals`**' - verifica dei segnali dei processi

SINTASSI

```
<kernel/proc.h>
void proc_sch_signals (void);
```

DESCRIZIONE

La funzione ***proc_sch_signals()*** ha il compito di scandire tutti i processi della tabella ***proc_table[]***, per verificare lo stato di attivazione dei segnali e procedere di conseguenza.

Dal punto di vista pratico, la funzione si limita a scandire i numeri PID possibili, demandando ad altre funzioni il compito di fare qualcosa nel caso fosse attivato l'indicatore di un segnale. Va comunque osservato che os16 si limita a gestire le azioni predefinite, pertanto si può soltanto attivare o inibire i segnali, salvo i casi in cui questi non possono essere mascherati.

Questa funzione viene usata soltanto da *proc_scheduler(9)* [99.8.11], ogni volta che ci si prepara allo scambio con un altro processo.

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_scheduler.c`' [100.9.11]

'`kernel/proc/proc_sch_signals.c`' [100.9.8]

VEDERE ANCHE

proc_sig_term(9) [99.8.19], *proc_sig_core(9)* [99.8.14], *proc_sig_chld(9)* [99.8.12], *proc_sig_cont(9)* [99.8.13], *proc_sig_stop(9)* [99.8.18].

99.8.9 os16: proc_sch_terminals(9)

NOME

'**proc_sch_terminals**' - acquisizione di un carattere dal terminale attivo

SINTASSI

```
<kernel/proc.h>
void proc_sch_terminals (void);
```

DESCRIZIONE

La funzione ***proc_sch_terminals()*** ha il compito di verificare la presenza di un carattere digitato dalla console. Se verifica che effettivamente è stato digitato un carattere, dopo aver determinato a quale terminale virtuale si riferisce, determina se per quel terminale era già stato accumulato un carattere, e se è effettivamente così, sovrascrive quel carattere ma annota anche che l'inserimento precedente è stato perduto.

Successivamente verifica se quel terminale virtuale è associato a un gruppo di processi; se è così e se il carattere corrisponde alla combinazione [*Ctrl c*], invia il segnale SIGINT a tutti i processi di quel gruppo, ma senza poi accumulare il carattere.

Indipendentemente dal fatto che il terminale appartenga a un gruppo di processi, controlla che il carattere inserito sia stato ottenuto, rispettivamente, con le combinazioni di tasti [*Ctrl q*], [*Ctrl r*], [*Ctrl s*] e [*Ctrl t*], nel qual caso attiva la console virtuale relativa (dalla prima alla quarta), evitando di accumulare il carattere.

Alla fine, scandisce tutti i processi sospesi in attesa di input dal terminale, risvegliandoli (ogni processo deve poi verificare se effettivamente c'è un carattere per sé oppure no, e se non c'è dovrebbe rimettersi in attesa).

Questa funzione viene usata soltanto da *proc_scheduler(9)* [99.8.11], ogni volta che ci si prepara allo scambio con un altro processo.

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_scheduler.c`' [100.9.11]

'`kernel/proc/proc_sch_terminals.c`' [100.9.9]

99.8.10 os16: proc_sch_timers(9)

«

NOME

'**proc_sch_timers**' - verifica dell'incremento del contatore del tempo

SINTASSI

```
<kernel/proc.h>
void proc_sch_timers (void);
```

DESCRIZIONE

La funzione ***proc_sch_timers()*** verifica che il calendario si sia incrementato di almeno una unità temporale (per os16 è un secondo soltanto) e se è così, va a risvegliare tutti i processi sospesi in attesa del passaggio di un certo tempo. Tali processi, una volta messi effettivamente in funzione, devono verificare che sia trascorsa effettivamente la quantità di tempo desiderata, altrimenti devono rimettersi a riposo in attesa del tempo rimanente.

Questa funzione viene usata soltanto da *proc_scheduler(9)* [99.8.11], ogni volta che ci si prepara allo scambio con un altro processo.

FILE SORGENTI

'kernel/proc.h' [100.9]

'kernel/proc/proc_scheduler.c' [100.9.11]

'kernel/proc/proc_sch_timers.c' [100.9.10]

99.8.11 os16: proc_scheduler(9)

«

NOME

'**proc_scheduler**' - schedulatore

SINTASSI

```
<kernel/proc.h>
void proc_scheduler (uint16_t *sp, segment_t *segment_d);
```

ARGOMENTI

Argomento	Descrizione
`extern uint16_t` ***_ksp*** `;`	L'indice della pila del kernel.
`uint16_t *`***sp***	Puntatore all'indice della pila del processo interrotto.
`segment_t *`***segment_d***	Puntatore al segmento dati del processo interrotto.

DESCRIZIONE

La funzione ***proc_scheduler()*** viene avviata a seguito di un'interruzione hardware, dovuta al temporizzatore, oppure a seguito di un'interruzione software, dovuta a una chiamata di sistema.

La funzione determina qual è il processo interrotto, scandendo la tabella dei processi alla ricerca di quello il cui segmento dati corrisponde al valore ***segment_d***. Per questo si avvale di *proc_find(9)* [99.8.5].

Successivamente verifica se ci sono processi in attesa di un evento del temporizzatore o del terminale, inoltre verifica se ci sono processi con segnali in attesa di essere presi in considerazione. per fare questo si avvale di *proc_sch_timers(9)* [99.8.10], *proc_sch_terminals(9)* [99.8.9] e *proc_sch_signals(9)* [99.8.8], che provvedono a fare ciò che serve in presenza degli eventi di propria competenza.

Si occupa quindi di annotare il tempo di CPU utilizzato dal processo appena sospeso, misurato in unità di tempo a cui si riferisce il tipo '**`clock_t`**'.

Successivamente scandisce la tabella dei processi alla ricerca di un altro processo da mettere in funzione, al posto di quello sospeso. Se trova un processo pronto per questo lo elegge a processo attivo, declassando quello sospeso a processo pronto ma in attesa, inoltre aggiorna i valori per le variabili ****sp*** e ****segment_d***.

Al termine salva nella variabile globale ***_ksp*** il valore dell'indice della pila del kernel, come appare nelle informazioni della tabella dei processi e poi manda il messaggio «EOI» (*end of interrupt* al «PIC 1» (*programmable interrupt controller*).

Questa funzione viene usata dalla routine *isr_1C(9)* [99.8.1] del file '`kernel/proc/_isr.s`' e dalla funzione *sysroutine(9)* [99.8.28].

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_scheduler.c`' [100.9.11]

VEDERE ANCHE

proc_find(9) [99.8.5], *proc_sch_timers(9)* [99.8.10], *proc_sch_signals(9)* [99.8.8], *proc_sch_terminals(9)* [99.8.9].

99.8.12 os16: proc_sig_chld(9)

«

NOME

'**`proc_sig_chld`**' - procedura associata alla ricezione di un segnale SIGCHLD

SINTASSI

```
<kernel/proc.h>
void proc_sig_chld (pid_t parent, int sig);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***parent***	Numero del processo considerato, il quale potrebbe avere ricevuto un segnale SIGCHLD.
`int` ***sig***	Numero del segnale: deve trattarsi esclusivamente di quanto corrispondente a SIGCHLD.

DESCRIZIONE

La funzione ***proc_sig_chld()*** si occupa di verificare che il processo specificato con il parametro ***parent*** abbia ricevuto precedentemente un segnale SIGCHLD. Se risulta effettivamente così, allora va a verificare se tale segnale risulta ignorato per quel processo: se è preso in considerazione verifica ancora se quel processo è sospeso proprio in attesa di un segnale SIGCHLD. Se si tratta di un processo che sta attendendo tale segnale, allora viene risvegliato, altrimenti, sempre ammesso che comunque il segnale non sia ignorato, la funzione elimina tutti i processi figli di ***parent***, i quali risultano già defunti, ma non ancora rimossi dalla tabella dei processi (pertanto processi «zombie»).

In pratica, se il processo ***parent*** sta attendendo un segnale SIGCHLD, significa che al risveglio si aspetta di verificare la morte di uno dei suoi processi figli, in modo da poter ottenere il valore di uscita con cui questo si è concluso. Diversamente, non c'è modo di informare il processo ***parent*** di tali conclusioni, per cui a nulla servirebbe continuare a mantenerne le tracce nella tabella dei processi.

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [99.8.8].

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_sig_chld.c`' [100.9.12]

VEDERE ANCHE

proc_sig_status(9) [99.8.17], *proc_sig_ignore(9)* [99.8.15], *proc_sig_off(9)* [99.8.16].

99.8.13 os16: proc_sig_cont(9)

NOME

'**`proc_sig_cont`**' - ripresa di un processo sospeso in attesa di qualcosa

SINTASSI

```
<kernel/proc.h>
void proc_sig_cont (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Numero del processo considerato.
`int` ***sig***	Numero del segnale: deve trattarsi esclusivamente di quanto corrispondente a SIGCONT.

DESCRIZIONE

La funzione ***proc_sig_cont()*** si occupa di verificare che il processo specificato con il parametro ***pid*** abbia ricevuto precedentemente un segnale SIGCONT e che questo non sia stato disabilitato. In tal caso, assegna al processo lo status di «pronto» ('**`PROC_READY`**'), ammesso che non si trovasse già in questa situazione.

Lo scopo del segnale SIGCONT è quindi quello di far riprendere un processo che in precedenza fosse stato sospeso attraverso un segnale SIGSTOP, SIGTSTP, SIGTTIN oppure SIGTTOU.

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [99.8.8].

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_sig_cont.c`' [100.9.13]

VEDERE ANCHE

proc_sig_status(9) [99.8.17], *proc_sig_ignore(9)* [99.8.15], *proc_sig_off(9)* [99.8.16].

99.8.14 os16: proc_sig_core(9)

«

NOME

'`proc_sig_core`' - chiusura di un processo e scarico della memoria su file

SINTASSI

```
<kernel/proc.h>
void proc_sig_core (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Numero del processo considerato.
`int` ***sig***	Numero del segnale: deve trattarsi di un segnale a cui si associa in modo predefinito la conclusione e lo scarico della memoria.

DESCRIZIONE

La funzione ***proc_sig_core()*** si occupa di verificare che il processo specificato con il parametro ***pid*** abbia ricevuto precedentemente un segnale tale da richiedere la conclusione e lo scarico della memoria del processo stesso, e che il segnale in questione non sia stato disabilitato. In tal caso, la funzione chiude il processo, ma prima ne scarica la memoria su uno o due file, avvalendosi per questo della funzione *proc_dump_memory(9)* [99.8.4].

Un segnale atto a produrre lo scarico della memoria, potrebbe essere prodotto anche a seguito di un errore rilevato dalla CPU, come una divisione per zero. Tuttavia, il kernel di os16 non riesce a intrappolare errori di questo tipo, dato che dalla tabella IVT vengono presi in considerazione soltanto l'impulso del temporizzatore e le chiamate di sistema. In altri termini, se un programma produce effettivamente un errore così grave da essere rilevato dalla CPU, al sistema operativo non arriva alcuna comunicazione. Pertanto, tali segnali possono essere soltanto provocati deliberatamente.

Lo scarico della memoria, nell'eventualità di un errore così grave, dovrebbe servire per consentire un'analisi dello stato del processo nel momento del verificarsi di un errore fatale. Sotto questo aspetto, va anche considerato che l'area dati dei processi è priva di etichette che possano agevolare l'interpretazione dei contenuti e, di conseguenza, non ci sono strumenti che consentano tale attività.

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [99.8.8].

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_sig_core.c`' [100.9.14]

VEDERE ANCHE

proc_sig_status(9) [99.8.17], *proc_sig_ignore(9)* [99.8.15], *proc_sig_off(9)* [99.8.16], *proc_dump_memory(9)* [99.8.4].

99.8.15 os16: proc_sig_ignore(9)

NOME

'**`proc_sig_ignore`**' - verifica dello stato di inibizione di un segnale

SINTASSI

```
<kernel/proc.h>
int proc_sig_ignore (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Numero del processo considerato.
`int` ***sig***	Numero del segnale da verificare.

DESCRIZIONE

La funzione ***proc_sig_ignore()*** verifica se, per un certo processo ***pid***, il segnale ***sig*** risulti inibito.

Questa funzione viene usata da *proc_sig_chld(9)* [99.8.12], *proc_sig_cont(9)* [99.8.13], *proc_sig_core(9)* [99.8.14], *proc_sig_stop(9)* [99.8.18] e *proc_sig_term(9)* [99.8.19], per verificare se un segnale sia stato inibito, prima di applicarne le conseguenze, nel caso fosse stato ricevuto.

VALORE RESTITUITO

Valore	Significato
1	Il segnale risulta bloccato (inibito).
0	Il segnale è abilitato regolarmente.

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_sig_ignore.c`' [100.9.15]

VEDERE ANCHE

proc_sig_chld(9) [99.8.12], *proc_sig_cont(9)* [99.8.13], *proc_sig_core(9)* [99.8.14], *proc_sig_stop(9)* [99.8.18]m *proc_sig_term(9)* [99.8.19].

99.8.16 os16: proc_sig_on(9)

NOME

'**`proc_sig_on`**', '**`proc_sig_off`**' - registrazione o cancellazione di un segnale per un processo

SINTASSI

```
<kernel/proc.h>
void proc_sig_on  (pid_t pid, int sig);
void proc_sig_off (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Numero del processo considerato.
`int` ***sig***	Numero del segnale da registrare o da cancellare.

DESCRIZIONE

La funzione ***proc_sig_on()*** annota per il processo ***pid*** la ricezione del segnale ***sig***; la funzione ***proc_sig_off()*** procede invece in senso opposto, cancellando quel segnale.

La funzione ***proc_sig_off()*** viene usata quando l'azione prevista per un segnale che risulta ricevuto è stata eseguita, allo scopo di riportare l'indicatore di quel segnale in una condizione di riposo. Si tratta delle funzioni *proc_sig_chld(9)* [99.8.12], *proc_sig_cont(9)* [99.8.13], *proc_sig_core(9)* [99.8.14], *proc_sig_stop(9)* [99.8.18] e *proc_sig_term(9)* [99.8.19].

La funzione ***proc_sig_on()*** viene usata quando risulta acquisito un segnale o quando il contesto lo deve produrre, per annotarlo. Si tratta delle funzioni *proc_sys_exit(9)* [99.8.21] e *proc_sys_kill(9)* [99.8.23].

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_sig_on.c`' [100.9.17]

'`kernel/proc/proc_sig_off.c`' [100.9.16]

VEDERE ANCHE

proc_sys_exit(9) [99.8.21], *proc_sys_kill(9)* [99.8.23], *proc_sig_chld(9)* [99.8.12], *proc_sig_cont(9)* [99.8.13], *proc_sig_core(9)* [99.8.14], *proc_sig_stop(9)* [99.8.18], *proc_sig_term(9)* [99.8.19].

99.8.17 os16: proc_sig_status(9)

NOME

'**proc_sig_status**' - verifica dello stato di ricezione di un segnale

SINTASSI

```
<kernel/proc.h>
int proc_sig_status (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t ***pid***	Numero del processo considerato.
int ***sig***	Numero del segnale da verificare.

DESCRIZIONE

La funzione ***proc_sig_status()*** verifica se, per un certo processo ***pid***, il segnale ***sig*** risulti essere stato ricevuto (registrato).

Questa funzione viene usata da *proc_sig_chld(9)* [99.8.12], *proc_sig_cont(9)* [99.8.13], *proc_sig_core(9)* [99.8.14], *proc_sig_stop(9)* [99.8.18] e *proc_sig_term(9)* [99.8.19], per verificare se un segnale è stato ricevuto effettivamente, prima di applicarne eventualmente le conseguenze.

VALORE RESTITUITO

Valore	Significato
1	Il segnale risulta ricevuto.
0	Il segnale risulta cancellato.

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_sig_status.c`' [100.9.18]

VEDERE ANCHE

proc_sig_chld(9) [99.8.12], *proc_sig_cont(9)* [99.8.13], *proc_sig_core(9)* [99.8.14], *proc_sig_stop(9)* [99.8.18], *proc_sig_term(9)* [99.8.19].

99.8.18 os16: proc_sig_stop(9)

NOME

'**proc_sig_stop**' - sospensione di un processo

SINTASSI

```
<kernel/proc.h>
void proc_sig_stop (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t ***pid***	Numero del processo considerato.
int ***sig***	Numero del segnale: deve trattarsi di SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU.

DESCRIZIONE

La funzione ***proc_sig_stop()*** si occupa di verificare che il processo specificato con il parametro ***pid*** abbia ricevuto precedentemente un segnale SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU, e che questo non sia stato disabilitato. In tal caso, sospende il processo, lasciandolo in attesa di un segnale (SIGCONT).

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [99.8.8].

FILE SORGENTI

'`kernel/proc.h`' [100.9]

'`kernel/proc/proc_sig_stop.c`' [100.9.19]

VEDERE ANCHE

proc_sig_status(9) [99.8.17], *proc_sig_ignore(9)* [99.8.15], *proc_sig_off(9)* [99.8.16].

99.8.19 os16: proc_sig_term(9)

NOME

'**proc_sig_term**' - conclusione di un processo

SINTASSI

```
<kernel/proc.h>
void proc_sig_term (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t *pid*	Numero del processo considerato.
int *sig*	Numero del segnale: deve trattarsi di un segnale per cui si associa la conclusione del processo, ma senza lo scarico della memoria.

DESCRIZIONE

La funzione ***proc_sig_term()*** si occupa di verificare che il processo specificato con il parametro ***pid*** abbia ricevuto precedentemente un segnale per cui si prevede generalmente la conclusione del processo. Inoltre, la funzione verifica che il segnale non sia stato inibito, con l'eccezione che per il segnale SIGKILL un'eventuale inibizione non viene considerata (in quanto segnale non mascherabile). Se il segnale risulta ricevuto e valido, procede con la conclusione del processo.

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [99.8.8].

FILE SORGENTI

'kernel/proc.h' [100.9]

'kernel/proc/proc_sig_term.c' [100.9.20]

VEDERE ANCHE

proc_sig_status(9) [99.8.17], *proc_sig_ignore(9)* [99.8.15], *proc_sig_off(9)* [99.8.16], *proc_sys_exit(9)* [99.8.21].

99.8.20 os16: proc_sys_exec(9)

«

NOME

'`proc_sys_exec`' - sostituzione di un processo esistente con un altro, ottenuto dal caricamento di un file eseguibile

SINTASSI

```
<kernel/proc.h>
int proc_sys_exec (uint16_t *sp, segment_t *segment_d,
                   pid_t pid, const char *path,
                   unsigned int argc, char *arg_data,
                   unsigned int envc, char *env_data);
```

ARGOMENTI

Argomento	Descrizione
`uint16_t *sp`	Puntatore alla variabile contenente l'indice della pila dei dati del processo che ha eseguito la chiamata di sistema *execve(2)* [93.10].
`segment_t *segment_d`	Puntatore alla variabile contenente il valore del segmento dati del processo che ha eseguito la chiamata di sistema *execve(2)* [93.10].
`pid_t pid`	Il numero del processo corrispondente.
`const char *path`	Il percorso assoluto del file da caricare ed eseguire.
`unsigned int argc`	La quantità di argomenti per l'avvio del nuovo processo, incluso il nome del processo stesso.
`char *arg_data`	Una sequenza di stringhe (dopo la terminazione di una inizia la successiva), ognuna contenente un argomento da passare al processo.
`unsigned int envc`	La quantità di variabili di ambiente da passare al nuovo processo.
`char *env_data`	Una sequenza di stringhe (dopo la terminazione di una inizia la successiva), ognuna contenente l'assegnamento di una variabile di ambiente.

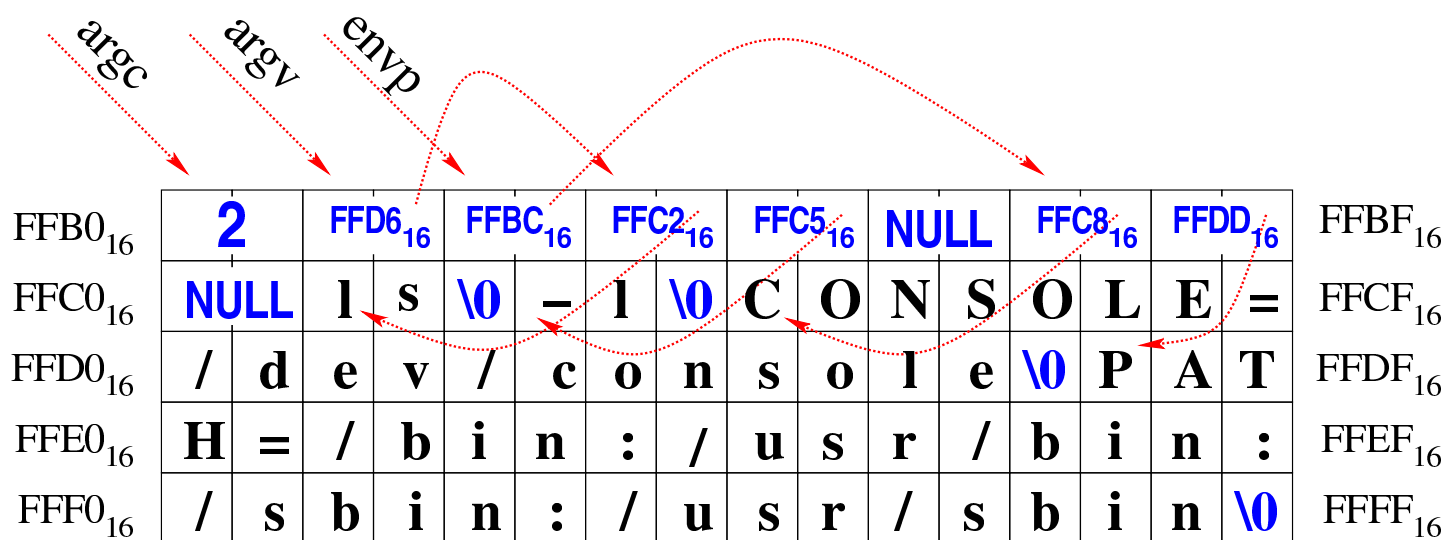
I parametri ***arg_data*** e ***env_data*** sono stringhe multiple, nel senso che sono separate le une dalle altre dal codice nullo di terminazione. Per sapere quante sono effettivamente le stringhe da cercare a partire dai puntatori che costituiscono effettivamente questi due parametri, si usano ***argc*** e ***envc***.

DESCRIZIONE

La funzione ***proc_sys_exec()*** serve a mettere in pratica la chiamata di sistema *execve(2)* [93.10], destinata a rimpiazzare il processo in corso con un nuovo processo, caricato da un file eseguibile.

La funzione ***proc_sys_exec()***, dopo aver verificato che si tratti effettivamente di un file eseguibile valido e che ci siano i permessi per metterlo in funzione, procede all'allocazione della memoria, dividendo se necessario l'area codice da quella dei dati, quindi legge il file e copia opportunamente le componenti di questo nelle aree di memoria allocate.

Terminato il caricamento del file, viene ricostruita in memoria la pila dei dati del nuovo processo. Prima si mettono sul fondo le stringhe delle variabili di ambiente e quelle degli argomenti della chiamata, quindi si aggiungono i puntatori alle stringhe delle variabili di ambiente, ricostruendo così l'array noto convenzionalmente come '**`envp[]`**', continuando con l'aggiunta dei puntatori alle stringhe degli argomenti della chiamata, per riprodurre l'array '**`argv[]`**'. Per ricostruire gli argomenti della chiamata della funzione ***main()*** dell'applicazione, vanno però aggiunti ancora: il puntatore all'inizio dell'array delle stringhe che descrivono le variabili di ambiente, il puntatore all'array delle stringhe che descrivono gli argomenti della chiamata e il valore che rappresenta la quantità di argomenti della chiamata.

Figura 99.145. Caricamento degli argomenti della chiamata della funzione ***main()***.

Fatto ciò, vanno aggiunti tutti i valori necessari allo scambio dei processi, costituiti dai vari registri da rimpiazzare.

Superato il problema della ricostruzione della pila dei dati, la funzione ***proc_sys_exec()*** predispone i descrittori di standard input, standard output e standard error, quindi libera la memoria usata dal processo chiamante e ne rimpiazza i dati nella tabella dei processi con quelli del nuovo processo caricato.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**SYS_EXEC**'.

FILE SORGENTI

'lib/unistd/execve.c' [101.17.13]

'lib/sys/os16/sys.s' [101.12.15]

'kernel/proc.h' [100.9]

'kernel/proc/_isr.s' [100.9.1]

'kernel/proc/sysroutine.c' [100.9.30]

'kernel/proc/proc_sys_exec.c' [100.9.21]

VEDERE ANCHE

execve(2) [93.10], *sys(2)* [93.37], *isr_80(9)* [99.8.1], *sysroutine(9)* [99.8.28], *proc_scheduler(9)* [99.8.11], *path_inode(9)* [99.3.36], *inode_check(9)* [99.3.16], *inode_put(9)* [99.3.24], *inode_file_read(9)* [99.3.18], *dev_io(9)* [99.1.1], *fd_close(9)* [99.3.3].

99.8.21 os16: proc_sys_exit(9)

«

NOME

'**proc_sys_exit**' - chiusura di un processo elaborativo

SINTASSI

```
<kernel/proc.h>
void proc_sys_exit (pid_t pid, int status);
```

ARGOMENTI

Argomento	Descrizione
pid_t ***pid***	Il numero del processo elaborativo da concludere.
int ***status***	Il valore di uscita del processo da concludere.

DESCRIZIONE

La funzione ***proc_sys_exit()*** conclude il processo indicato come argomento, chiudendo tutti i descrittori di file che risultano ancora aperti e liberando la memoria. Precisamente compie i passaggi seguenti:

- aggiorna la tabella dei processi indicando per questo lo stato di «zombie» e annotando il valore di uscita;
- chiude i descrittori di file che risultano aperti;
- chiude l'inode della directory corrente;
- se si tratta del processo principale di un gruppo di processi, allora chiude anche il terminale di controllo;
- libera la memoria utilizzata dal processo, verificando comunque che l'area usata per il codice non sia abbinata anche a un altro processo, nel qual caso l'area del codice verrebbe preservata;
- se ci sono dei processi figli di quello che si va a chiudere, questi vengono abbandonati e affidati al processo numero uno ('**`init`**');
- se sono stati abbandonati dei processi, invia il segnale SIGCHLD al processo numero uno ('**`init`**');
- invia al processo genitore il segnale, in modo che possa valutare, eventualmente, il valore di uscita del processo ormai defunto.

Questa funzione viene usata principalmente da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_EXIT`**', e anche dalle funzioni *proc_sig_core(9)* [99.8.14] e *proc_sig_term(9)* [99.8.19].

FILE SORGENTI

'`lib/unistd/_exit.c`' [101.17.1]

'`lib/stdlib/_Exit.c`' [101.10.1]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_exit.c`' [100.9.22]

VEDERE ANCHE

_exit(2) [93.2], *sys(2)* [93.37], *isr_80(9)* [99.8.1], *sysroutine(9)* [99.8.28], *proc_scheduler(9)* [99.8.11], *proc_sig_core(9)* [99.8.14], *proc_sig_term(9)* [99.8.19], *fd_close(9)* [99.3.3], *inode_put(9)* [99.3.24], *proc_sig_on(9)* [99.8.16].

99.8.22 os16: proc_sys_fork(9)

NOME

'**proc_sys_fork**' - sdoppiamento di un processo elaborativo

SINTASSI

```
<kernel/proc.h>
pid_t proc_sys_fork (pid_t ppid, uint16_t sp);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***ppid***	Il numero del processo che chiede di creare un figlio uguale a se stesso.
`uint16_t` ***sp***	Indice della pila del processo da duplicare.

DESCRIZIONE

La funzione ***proc_sys_fork()*** crea un duplicato del processo chiamante, il quale diventa figlio dello stesso. Precisamente, la funzione compie i passaggi seguenti:

- cerca un alloggiamento libero nella tabella dei processi e procede solo se questo risulta disponibile effettivamente;
- per sicurezza, analizza i processi che risultano essere defunti (zombie) e ne inizializza i valori delle allocazioni in memoria;
- alloca la memoria necessaria a ottenere la copia del processo, tenendo conto che se il processo originario divide l'area codice da quella dei dati, è necessario allocare soltanto lo spazio per l'area dati, in quanto quella del codice può essere condivisa (essendo usata soltanto in lettura);
- compila le informazioni necessarie nella tabella dei processi, relative al nuovo processo da produrre, dichiarandolo come figlio di quello chiamante;
- incrementa il contatore di utilizzo dell'inode che rappresenta la directory corrente, in quanto un nuovo processo la va a utilizzare;
- duplica i descrittori di file già aperti per il processo da duplicare, incrementando di conseguenza il contatore dei riferimenti nella tabella dei file;
- modifica i valori dei registri di segmento nella pila dei dati riferita al processo nuovo, per renderli coerenti con la nuova collocazione in memoria;

- mette il nuovo processo nello stato di pronto, annotandolo così nella tabella dei processi;
- restituisce il numero del nuovo processo: nel processo figlio, invece, non restituisce alcunché.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_FORK`**'.

VALORE RESTITUITO

La funzione restituisce al processo chiamante il numero del processo figlio, mentre il risultato che si ottiene nel processo figlio che si trova a riprendere il funzionamento dallo stesso punto, è semplicemente zero. Ciò consente di distinguere quale sia il processo genitore e quale è invece il figlio. Se la funzione non è in grado di portare a termine il lavoro di duplicazione dei processi, restituisce il valore -1, aggiornando di conseguenza la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`ENOMEM`	Non c'è memoria sufficiente, oppure la tabella dei processi è occupata completamente.

FILE SORGENTI

'`lib/unistd/fork.c`' [101.17.17]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_fork.c`' [100.9.23]

VEDERE ANCHE

fork(2) [93.14], *sys(2)* [93.37], *isr_80(9)* [99.8.1], *sysroutine(9)* [99.8.28], *proc_scheduler(9)* [99.8.11], *dev_io(9)* [99.1.1].

99.8.23 os16: proc_sys_kill(9)

NOME

'**`proc_sys_kill`**' - invio di un segnale a uno o più processi elaborativi

SINTASSI

```
<kernel/proc.h>
int proc_sys_kill (pid_t pid_killer, pid_t pid_target, int sig);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid_killer***	Il numero del processo per conto del quale si invia il segnale.
`pid_t` ***pid_target***	Il numero del processo che dovrebbe ricevere il segnale.
`int` ***sig***	Il numero del segnale da inviare.

DESCRIZIONE

La funzione ***proc_sys_kill()*** invia il segnale ***sig*** al processo numero ***pid_target***, ammesso che il processo ***pid_killer*** abbia i privilegi necessari a farlo. Tuttavia, se il numero ***pid_target*** è zero o -1, si richiede alla funzione l'invio del segnale a un insieme di processi. La tabella successiva descrive i vari casi.

Identità efficace del processo ***pid_killer***	Valore di ***pid_target***	Effetto.
--	< -1	Il valore di ***pid_target*** non è ammissibile: si ottiene un errore.
0	-1	Viene inviato il segnale ***sig*** a tutti i processi con UID ≥ 2 (si esclude il kernel e il processo numero uno, '**`init`**').
> 0	-1	Viene inviato il segnale ***sig*** a tutti i processi con UID ≥ 1 (si esclude il kernel) la cui identità efficace coincide con quella di ***pid_killer***.
--	0	Viene inviato il segnale ***sig*** a tutti i processi che appartengono allo stesso gruppo di ***pid_target***.
--	> 0	Viene inviato il segnale ***sig*** al processo ***pid_target***, purché l'identità reale o efficace del processo ***pid_killer*** sia uguale all'identità reale o salvata del processo ***pid_target***.

Si osservi che il preteso invio di un segnale pari a zero, ovvero di un segnale nullo, non produce alcun effetto, ma la funzione segnala comunque di avere completato l'operazione con successo.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_KILL`**'.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Operazione fallita, con aggiornamento della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`ESRCH`	Il processo ***pid_target*** non esiste, non è un processo che possa ricevere segnali, oppure il valore dato non è interpretabile in alcun modo.
`EPERM`	Il processo ***pid_killer*** non ha i privilegi necessari a inviare il segnale a ***pid_target***.

FILE SORGENTI

'`lib/signal/kill.c`' [101.8.1]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_kill.c`' [100.9.24]

VEDERE ANCHE

kill(2) [93.22], *sys(2)* [93.37], *isr_80(9)* [99.8.1], *sysroutine(9)* [99.8.28], *proc_scheduler(9)* [99.8.11], *proc_sig_on(9)* [99.8.16].

99.8.24 os16: proc_sys_seteuid(9)

NOME

'**`proc_sys_seteuid`**' - modifica dell'identità efficace

SINTASSI

```
<kernel/proc.h>
int proc_sys_seteuid (pid_t pid, uid_t euid);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Processo su cui intervenire per il cambiamento di identità efficace.
`uid_t` ***euid***	Nuova identità efficace richiesta.

DESCRIZIONE

La funzione ***proc_sys_seteuid()*** modifica l'identità efficace del processo ***pid***, purché si verifichino certe condizioni:

- se il processo ***pid*** è zero, l'identità efficace viene modificata senza altre verifiche;
- se l'identità efficace che ha già il processo coincide con quella nuova richiesta, non viene apportata alcuna modifica (per ovvi motivi);
- se la nuova identità efficace corrisponde all'identità reale del processo, oppure se corrisponde alla sua identità salvata, allora la modifica di quella efficace ha luogo come richiesto;
- in tutti gli altri casi si ottiene un errore.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_SETEUID`**'.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Operazione fallita, con aggiornamento della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EPERM`	Il processo ***pid*** non può cambiare l'identità efficace con il valore richiesto.

FILE SORGENTI

'`lib/unistd/seteuid.c`' [101.17.30]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_seteuid.c`' [100.9.25]

VEDERE ANCHE

seteuid(2) [93.33], *sys(2)* [93.37], *isr_80(9)* [99.8.1], *sysroutine(9)* [99.8.28], *proc_scheduler(9)* [99.8.11], *proc_sys_setuid(9)* [99.8.25].

99.8.25 os16: proc_sys_setuid(9)

«

NOME

'**`proc_sys_setuid`**' - modifica dell'identità

SINTASSI

```
<kernel/proc.h>
int proc_sys_setuid (pid_t pid, uid_t uid);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Processo su cui intervenire per il cambiamento di identità.
`uid_t` ***uid***	Nuova identità richiesta.

DESCRIZIONE

La funzione ***proc_sys_setuid()*** modifica l'identità del processo ***pid***, oppure tutti i tipi di identità, a seconda di certe condizioni:

- se l'identità efficace del processo ***pid*** è zero, viene modificata l'identità reale, quella salvata e quella efficace, utilizzando il nuovo valore ***uid***;
- se l'identità efficace del processo coincide già con quella del valore richiesto ***uid***, non viene apportata alcuna modifica e la funzione si conclude con successo;
- se l'identità reale o quella salvato del processo ***pid*** coincide con l'identità richiesta ***uid***, allora viene modificata l'identità efficace del processo con il valore ***uid***;
- in tutti gli altri casi si ottiene un errore.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_SETUID`**'.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
−1	Operazione fallita, con aggiornamento della variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EPERM`	Il processo ***pid*** non può cambiare identità come richiesto.

FILE SORGENTI

'`lib/unistd/setuid.c`' [101.17.32]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_setuid.c`' [100.9.26]

VEDERE ANCHE

setuid(2) [93.33], *sys(2)* [93.37], *isr_80(9)* [99.8.1], *sysroutine(9)* [99.8.28], *proc_scheduler(9)* [99.8.11], *proc_sys_seteuid(9)* [99.8.24].

99.8.26 os16: proc_sys_signal(9)

«

NOME

'**`proc_sys_signal`**' - modifica della configurazione dei segnali

SINTASSI

```
<kernel/proc.h>
sighandler_t proc_sys_signal (pid_t pid, int sig,
                              sighandler_t handler);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Processo su cui intervenire per il cambiamento di configurazione.
`int` ***sig***	Segnale da riconfigurare.
`sighandler_t` ***handler***	Nuova azione da associare al segnale. Si possono solo usare i valori corrispondenti a '**`SIG_IGN`**' e '**`SIG_DFL`**', con cui, rispettivamente, si inibisce il segnale o gli si attribuisce l'azione predefinita.

DESCRIZIONE

La funzione ***proc_sys_signal()*** ha il compito di modificare il comportamento del processo nel caso fosse ricevuto il segnale specificato. Teoricamente, il parametro ***handler*** potrebbe riferirsi a una funzione da eseguire allo scattare del segnale; tuttavia, os16 non è in grado di gestire questa evenienza e per ***handler*** si può specificare soltanto il valore corrispondente all'azione predefinita o a quella di inibizione del segnale.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '**`SYS_SIGNAL`**'.

VALORE RESTITUITO

La funzione restituisce il valore di ***handler*** abbinato precedentemente al processo. Se si verifica un errore, restituisce '**`SIG_ERR`**' e aggiorna la variabile ***errno*** del kernel.

ERRORI

Valore di ***errno***	Significato
`EINVAL`	La combinazione degli argomenti non è valida.

FILE SORGENTI

'`lib/signal/signal.c`' [101.8.2]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_signal.c`' [100.9.27]

VEDERE ANCHE

signal(2) [93.34], *sys(2)* [93.37], *isr_80(9)* [99.8.1], *sysroutine(9)* [99.8.28], *proc_scheduler(9)* [99.8.11], *proc_sys_kill(9)* [99.8.23].

99.8.27 os16: proc_sys_wait(9)

NOME

'`proc_sys_wait`' - attesa per la morte di un processo figlio

SINTASSI

```
<kernel/proc.h>
pid_t proc_sys_wait (pid_t pid, int *status);
```

ARGOMENTI

Argomento	Descrizione
`pid_t` ***pid***	Il processo che intende mettersi in attesa della morte di un proprio figlio.
`int *`***status***	Puntatore a una variabile atta a contenere il valore di uscita di un processo figlio defunto.

DESCRIZIONE

La funzione ***proc_sys_wait()*** ha il compito di mettere il processo ***pid*** in pausa, fino alla morte di uno dei propri processi figli.

Per realizzare questo compito, la funzione scandisce inizialmente la tabella dei processi alla ricerca di figli di ***pid***. Se tra questi ne esiste già uno defunto, allora aggiorna ****status*** con il valore di uscita di quello, liberando definitivamente la tabella dei processi dalle tracce di questo figlio. Se invece, pur avendo trovato dei figli, questi risultano ancora tutti in funzione, mette il processo ***pid*** in pausa, in attesa di un segnale SIGCHLD.

Questa funzione viene usata soltanto da *sysroutine(9)* [99.8.28], in occasione del ricevimento di una chiamata di sistema di tipo '`SYS_WAIT`'.

VALORE RESTITUITO

La funzione restituisce il numero PID del processo defunto, se c'è, aggiornando anche ****status*** con il valore di uscita dello stesso processo. Se invece il processo ***pid*** è stato messo in attesa, allora restituisce zero, mentre se non ci sono proprio figli di ***pid***, restituisce −1 e aggiorna la variabile ***errno*** del kernel.

ERRORI

Valore di *errno*	Significato
`ECHILD`	Non ci sono figli del processo ***pid*** e a nulla servirebbe attendere.

FILE SORGENTI

'`lib/sys/wait/wait.c`' [101.15.1]

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

'`kernel/proc/proc_sys_wait.c`' [100.9.28]

VEDERE ANCHE

wait(2) [93.43], *sys(2)* [93.37], *isr_80(9)* [99.8.1], *sysroutine(9)* [99.8.28], *proc_available(9)* [99.8.3], *proc_scheduler(9)* [99.8.11], *proc_sys_fork(9)* [99.8.22], *proc_sys_kill(9)* [99.8.23].

99.8.28 os16: sysroutine(9)

NOME

'**`sysroutine`**' - attuazione delle chiamate di sistema

SINTASSI

```
<kernel/proc.h>
void sysroutine (uint16_t *sp, segment_t *segment_d,
                 uint16_t syscallnr,
                 uint16_t msg_off, uint16_t msg_size);
```

ARGOMENTI

Argomento	Descrizione
`uint16_t *`***sp***	Puntatore all'indice della pila dei dati del processo che ha emesso la chiamata di sistema.
`segment_t *`***segment_d***	Puntatore al valore del segmento dati del processo che ha emesso la chiamata di sistema.
`uint16_t` ***syscallnr***	Il numero della chiamata di sistema.
`uint16_t` ***msg_off***	Nonostante il tipo di variabile, si tratta del **puntatore** alla posizione di memoria in cui inizia il messaggio con gli argomenti della chiamata di sistema, ma tale puntatore è valido solo nell'ambito del segmento ****segment_d***.
`uint16_t` ***msg_size***	La lunghezza del messaggio della chiamata di sistema.

DESCRIZIONE

La funzione ***sysroutine()*** viene chiamata esclusivamente dalla routine *isr_80(9)* [99.8.1], a seguito di una chiamata di sistema.

Inizialmente, la funzione individua il processo elaborativo corrispondente a quello che utilizza il segmento dati ****segment_d*** e l'**indirizzo efficace** dell'area di memoria contenente il messaggio della chiamata di sistema, traducendo le informazioni contenute in ***msg_off*** e ****segment_d***.

Attraverso un'unione di variabili strutturate, tutti i tipi di messaggi gestibili per le chiamate di sistema vengono dichiarati assieme in un'unica area di memoria. Successivamente, la funzione deve trasferire il messaggio, dall'indirizzo efficace calcolato precedentemente all'inizio dell'unione in questione.

Quando la funzione è in grado di accedere ai dati del messaggio, procede con una grande struttura di selezione, sulla base del tipo di messaggio, quindi esegue ciò che è richiesto, avvalendosi prevalentemente di altre funzioni, interpretando il messaggio in modo diverso a seconda del tipo di chiamata.

Il messaggio viene poi sovrascritto con le informazioni prodotte dall'azione richiesta, in particolare viene trasferito anche il valore della variabile ***errno*** del kernel, in modo che possa essere recepita anche dal processo che ha eseguito la chiamata, in caso di esito erroneo. Pertanto, il messaggio viene anche riscritto a partire dall'indirizzo efficace da cui era stato copiato precedentemente, in modo da renderlo disponibile effettivamente al processo chiamante.

Quando la funzione ***sysroutine()*** ha finito il suo lavoro, chiama a sua volta *proc_scheduler(9)* [99.8.11], perché con l'occasione provveda eventualmente alla

sostituzione del processo attivo con un altro che si trovi nello stato di pronto.

VALORE RESTITUITO

La funzione non restituisce alcun valore, in quanto tutto ciò che c'è da restituire viene trasmesso con la riscrittura del messaggio, nell'area di memoria originale.

FILE SORGENTI

'`lib/sys/os16/sys.s`' [101.12.15]

'`kernel/proc.h`' [100.9]

'`kernel/proc/_isr.s`' [100.9.1]

'`kernel/proc/sysroutine.c`' [100.9.30]

VEDERE ANCHE

sys(2) [93.37], *isr_80(9)* [99.8.1], *proc_scheduler(9)* [99.8.11], *dev_io(9)* [99.1.1], *path_chdir(9)* [99.3.30], *path_chmod(9)* [99.3.31], *path_chown(9)* [99.3.32], *fd_close(9)* [99.3.3], *fd_dup(9)* [99.3.4], *fd_dup2(9)* [99.3.4], *proc_sys_exec(9)* [99.8.20], *proc_sys_exit(9)* [99.8.21], *fd_chmod(9)* [99.3.1], *fd_chown(9)* [99.3.2], *fd_fcntl(9)* [99.3.6], *proc_sys_fork(9)* [99.8.22], *fd_stat(9)* [99.3.50], *proc_sys_kill(9)* [99.8.23], *path_link(9)* [99.3.38], *fd_lseek(9)* [99.3.7], *path_mkdir(9)* [99.3.39], *path_mknod(9)* [99.3.40], *path_mount(9)* [99.3.41], *fd_open(9)* [99.3.8], *fd_read(9)* [99.3.9], *proc_sys_seteuid(9)* [99.8.24], *proc_sys_setuid(9)* [99.8.25], *proc_sys_signal(9)* [99.8.26], *path_stat(9)* [99.3.50], *path_umount(9)* [99.3.41], *path_unlink(9)* [99.3.44], *proc_sys_wait(9)* [99.8.27], *fd_write(9)* [99.3.12].

99.9 os16: tty(9)

Il file '`kernel/tty.h`' [100.10] descrive le funzioni per la gestione dei terminali virtuali.

Per la descrizione dell'organizzazione della gestione dei terminali virtuali di os16, si rimanda alla sezione 90.5. La tabella successiva che sintetizza l'uso delle funzioni di questo gruppo, è tratta da quel capitolo.

Tabella 90.37. Funzioni per la gestione dei terminali, dichiarate nel file di intestazione '`kernel/tty.h`'.

Funzione	Descrizione
`void tty_init (void);`	Inizializza la gestione dei terminali. Viene usata una volta sola nella funzione ***main()*** del kernel.

Funzione	Descrizione
`tty_t *tty_reference (dev_t` ***device***`);`	Restituisce il puntatore a un elemento della tabella dei terminali. Se come numero di dispositivo si indica lo zero, si ottiene il riferimento a tutta la tabella; se non viene trovato il numero di dispositivo cercato, si ottiene il puntatore nullo.
`dev_t tty_console (dev_t` ***device***`);`	Seleziona la console indicata attraverso il numero di dispositivo che costituisce l'unico parametro. Se viene dato un valore a zero, si ottiene solo di conoscere qual è la console attiva. La console selezionata viene anche memorizzata in una variabile statica, per le chiamate successive della funzione. Se viene indicato un numero di dispositivo non valido, si seleziona implicitamente la prima console.
`int tty_read (dev_t` ***device***`);`	Legge un carattere dal terminale specificato attraverso il numero di dispositivo. Per la precisione, il carattere viene tratto dal campo relativo contenuto nella tabella dei terminali. Il carattere viene restituito dalla funzione come valore intero comune; se si ottiene zero significa che non è disponibile alcun carattere.
`void tty_write (dev_t` ***device***`, int` ***c***`);`	Scrive sullo schermo del terminale rappresentato dal numero di dispositivo, il carattere fornito come secondo parametro.

Codice

Capitolo 100

Script e sorgenti del kernel

100.1 os16: directory principale

100.1.1 bochs

Si veda la sezione 97.2.

```
10001 #!/bin/sh
10002
10003 bochs -q "boot:floppy" \
10004          "floppya: 1_44=floppy.a, status=inserted" \
10005          "floppyb: 1_44=floppy.b, status=inserted" \
10006          "keyboard_mapping: enabled=1, \
10007           map=/usr/share/bochs/keymaps/x11-pc-it.map" \
10008          "keyboard_type: xt" \
10009          "vga: none" \
10010          "romimage: file=\"/usr/share/bochs/BIOS-bochs-legacy\"" \
10011          "megs:1"
```

100.1.2 qemu

Si veda la sezione 97.2.

```
20001 #!/bin/sh
20002
20003 qemu -fda floppy.a \
20004      -fdb floppy.b \
20005      -boot order=a
20006
```

100.1.3 makeit

Si veda la sezione 97.2.

```
30001 #!/bin/sh
30002 #
30003 # makeit...
30004 #
30005 OPTION="$1"
30006 OS16PATH=""
30007 #
30008 edition () {
30009     local EDITION="kernel/main/build.h"
30010     echo -n                              > $EDITION
30011     echo -n "#define BUILD_DATE \""     >> $EDITION
30012     echo -n `date "+%Y.%m.%d %H:%M:%S"` >> $EDITION
30013     echo  "\""                          >> $EDITION
30014 }
30015 #
30016 #
30017 #
30018 makefile () {
30019     #
30020     local MAKEFILE="Makefile"
30021     local TAB="  "
30022     #
30023     local SOURCE_C=""
30024     local C=""
30025     local SOURCE_S=""
30026     local S=""
30027     #
30028     local c
30029     local s
30030     #
30031     # Trova i file in C.
30032     #
30033     for c in *.c
30034     do
30035         if [ -f $c ]
30036         then
30037         C=`basename $c .c`
30038             SOURCE_C="$SOURCE_C $C"
30039         fi
30040     done
30041     #
30042     # Trova i file in ASM.
30043     #
30044     for s in *.s
30045     do
```

```
30046           if [ -f $s ]
30047           then
30048           S=`basename $s .s`
30049               SOURCE_S="$SOURCE_S $S"
30050           fi
30051       done
30052       #
30053       # Prepara il file make.
30054       # GCC viene usato per potenziare il controllo degli errori.
30055       #
30056       echo -n                                                  > $MAKEFILE
30057       echo "# This file was made automatically"               >> $MAKEFILE
30058       echo "# by the script \`makeit\', based on the"         >> $MAKEFILE
30059       echo "# directory content."                             >> $MAKEFILE
30060       echo "# Please use \`makeit\' to compile and"           >> $MAKEFILE
30061       echo "# \`makeit clean\' to clean directories."         >> $MAKEFILE
30062       echo "#"                                                >> $MAKEFILE
30063       echo "c = $SOURCE_C"                                    >> $MAKEFILE
30064       echo "#"                                                >> $MAKEFILE
30065       echo "s = $SOURCE_S"                                    >> $MAKEFILE
30066       echo "#"                                                >> $MAKEFILE
30067       echo "all: \$(s) \$(c)"                                 >> $MAKEFILE
30068       echo "#"                                                >> $MAKEFILE
30069       echo "clean:"                                           >> $MAKEFILE
30070       echo "${TAB}@rm \$(c) \$(s) *.o *.assembler 2> /dev/null ; true" \
30071                                                               >> $MAKEFILE
30072       echo "${TAB}@rm *.symbols 2> /dev/null     ; true"      >> $MAKEFILE
30073       echo "${TAB}@pwd"                                       >> $MAKEFILE
30074       echo "#"                                                >> $MAKEFILE
30075       echo "\$(c):"                                           >> $MAKEFILE
30076       echo "${TAB}@echo \$@.c"                                >> $MAKEFILE
30077       echo "${TAB}@gcc -Wall -c -o \$@.o " \
30078                              "-I " \
30079                              "-I. " \
30080                              "-I$OS16PATH/lib " \
30081                              "-I$OS16PATH/ " \
30082                              "\$@.c"                          >> $MAKEFILE
30083       echo "${TAB}@rm \$@.o"                                  >> $MAKEFILE
30084       echo "${TAB}@bcc -ansi -0 -Mc -S -o \$@.assembler " \
30085                              "-I " \
30086                              "-I. " \
30087                              "-I$OS16PATH/lib " \
30088                              "-I$OS16PATH/ " \
30089                              "\$@.c"                          >> $MAKEFILE
30090       echo "${TAB}@bcc -ansi -0 -Mc -c -o \$@.o " \
30091                              "-I " \
30092                              "-I. " \
30093                              "-I$OS16PATH/lib " \
```

```
30094                           "-I$OS16PATH/ " \
30095                           "\$@.c"                                 >> $MAKEFILE
30096     echo "#"                                                      >> $MAKEFILE
30097     echo "\$(s):"                                                 >> $MAKEFILE
30098     echo "${TAB}@echo \$@.s"                                      >> $MAKEFILE
30099     echo "${TAB}@as86 -u -0 -o \$@.o -s \$@.symbols \$@.s"  >> $MAKEFILE
30100     #
30101 }
30102 #
30103 #
30104 #
30105 main () {
30106     #
30107     local CURDIR=`pwd`
30108     local OBJECTS
30109     local OBJLIB
30110     local EXEC
30111     local BASENAME
30112     local PROGNAME
30113     local d
30114     local c
30115     local s
30116     local o
30117     #
30118     edition
30119     #
30120     # Copia dello scheletro
30121     #
30122     if [ "$OPTION" = "clean" ]
30123     then
30124         #
30125         # La copia non va fatta.
30126         #
30127         true
30128     else
30129         cp -dpRv skel/etc   /mnt/os16.a/
30130         cp -dpRv skel/dev   /mnt/os16.a/
30131         mkdir               /mnt/os16.a/mnt/
30132         mkdir               /mnt/os16.a/tmp/
30133         chmod 0777          /mnt/os16.a/tmp/
30134         mkdir               /mnt/os16.a/usr/
30135         cp -dpRv skel/root  /mnt/os16.a/
30136         cp -dpRv skel/home  /mnt/os16.a/
30137         cp -dpRv skel/usr/* /mnt/os16.b/
30138     fi
30139     #
30140     #
30141     #
```

```
30142      for d in `find kernel`  \
30143               `find lib`     \
30144               `find applic`  \
30145               `find ported`
30146      do
30147          if [ -d "$d" ]
30148          then
30149              #
30150              # Sono presenti dei file C o ASM?
30151              #
30152              c=`echo $d/*.c | sed "s/ .*//"`
30153              s=`echo $d/*.s | sed "s/ .*//"`
30154              #
30155              if [ -f "$c" ] || [ -f "$s" ]
30156              then
30157                  #
30158                  # Sì
30159                  #
30160                  CURDIR=`pwd`
30161                  cd $d
30162                  #
30163                  # Ricrea il file make
30164                  #
30165                  makefile
30166                  #
30167                  # Pulisce quindi la directory
30168                  #
30169                  make clean
30170                  #
30171                  #
30172                  #
30173                  if [ "$OPTION" = "clean" ]
30174                  then
30175                      #
30176                      # È stata richiesta la pulitura, ma questa
30177                      # è appena stata fatta!
30178                      #
30179                      true
30180                  else
30181                      #
30182                      # Qualunque altro argomento viene considerato
30183                      # un 'make'.
30184                      #
30185                      if ! make
30186                      then
30187                          #
30188                          # La compilazione è fallita.
30189                          #
```

```
30190                        cd "$CURDIR"
30191                        exit
30192                    fi
30193                fi
30194                cd "$CURDIR"
30195            fi
30196        fi
30197    done
30198    #
30199    cd "$CURDIR"
30200    #
30201    # Link
30202    #
30203    if [ "$OPTION" = "clean" ]
30204    then
30205        #
30206        # Il collegamento non va fatto.
30207        #
30208        true
30209    else
30210        #
30211        # Collegamento dei file del kernel.
30212        #
30213        OBJECTS=""
30214        #
30215        for o in `find kernel   -name \*.o -print` \
30216                 `find lib      -name \*.o -print`
30217        do
30218            if    [ "$o" = "./kernel/main/crt0.o" ] \
30219               || [ "$o" = "./kernel/main/main.o" ] \
30220               || [ ! -e "$o" ]
30221            then
30222                true
30223            else
30224                OBJECTS="$OBJECTS $o"
30225            fi
30226        done
30227        #
30228        echo "Link"
30229        #
30230        ld86 -i -d -s -m -o kimage    \
30231                kernel/main/crt0.o    \
30232                kernel/main/main.o    \
30233                $OBJECTS
30234        #
30235        # Copia il kernel nel dischetto.
30236        #
30237        if mount | grep /mnt/os16.a > /dev/null
```

```
30238          then
30239              cp -f kimage /mnt/os16.a/boot
30240          else
30241              echo "[$0] Cannot copy the kernel image "
30242              echo "[$0]   inside the floppy disk image!"
30243          fi
30244          sync
30245          #
30246          # Collegamento delle applicazioni di os16.
30247          #
30248          OBJLIB=""
30249          #
30250          for o in `find lib      -name \*.o -print`
30251          do
30252              OBJLIB="$OBJLIB $o"
30253          done
30254          #
30255          # Scansione delle applicazioni interne.
30256          #
30257          for o in `find applic     -name \*.o -print`
30258          do
30259              if    [ "$o" = "applic/crt0.o" ] \
30260                 || [ ! -e "$o" ] \
30261                 || echo "$o" | grep ".crt0.o$" > /dev/null
30262              then
30263                  #
30264                  # Il file non esiste oppure si tratta di '...crt0.s'.
30265                  #
30266                  true
30267              else
30268                  #
30269                  # File oggetto differente da '...crt0.s'.
30270                  #
30271                  EXEC=`echo "$o" | sed "s/\.o$//"`
30272                  BASENAME=`basename $o .o`
30273                  if [ -e "applic/$BASENAME.crt0.o" ]
30274                  then
30275                      #
30276                      # Qui c'è un file '...crt0.o' specifico.
30277                      #
30278                      ld86 -i -d -s -o $EXEC \
30279                           applic/$BASENAME.crt0.o $o $OBJLIB
30280                  else
30281                      #
30282                      # Qui si usa il file 'crt0.o' generale.
30283                      #
30284                      ld86 -i -d -s -o $EXEC applic/crt0.o $o $OBJLIB
30285                  fi
```

```
30286                   #
30287                   if [ -x "applic/$BASENAME" ]
30288                   then
30289                       if mount | grep /mnt/os16.a > /dev/null
30290                       then
30291                           mkdir /mnt/os16.a/bin/ 2> /dev/null
30292                           cp -f "$EXEC" /mnt/os16.a/bin
30293                       else
30294                           echo "[$0] Cannot copy the application "
30295                           echo "[$0]   $BASENAME inside the floppy "
30296                           echo "[$0]   disk image!"
30297                           break
30298                       fi
30299                   fi
30300               fi
30301           done
30302           sync
30303           #
30304           # Collegamento delle applicazioni più semplici,
30305           # provenienti da altri sistemi operativi.
30306           #
30307           for o in `find ported/mix -name \*.o -print`
30308           do
30309               if    [ "$o" = "ported/mix/crt0.o" ] \
30310                  || [ ! -e "$o" ] \
30311                  || echo "$o" | grep ".crt0.o$" > /dev/null
30312               then
30313                   #
30314                   # Il file non esiste oppure si tratta di `...crt0.s'.
30315                   #
30316                   true
30317               else
30318                   #
30319                   # File oggetto differente da `...crt0.s'.
30320                   #
30321                   EXEC=`echo "$o" | sed "s/\.o$//"`
30322                   BASENAME=`basename $o .o`
30323                   if [ -e "ported/mix/$BASENAME.crt0.o" ]
30324                   then
30325                       #
30326                       # Qui c'è un file `...crt0.o' specifico.
30327                       #
30328                       ld86 -i -d -s -o $EXEC \
30329                            applic/$BASENAME.crt0.o $o $OBJLIB
30330                   else
30331                       #
30332                       # Qui si usa il file `crt0.o' generale.
30333                       #
```

```
30334                    ld86 -i -d -s -o $EXEC applic/crt0.o $o $OBJLIB
30335                fi
30336                #
30337                if [ -x "$EXEC" ]
30338                then
30339                    if mount | grep /mnt/os16.a > /dev/null
30340                    then
30341                        mkdir /mnt/os16.b/bin/ 2> /dev/null
30342                        cp -f "$EXEC" /mnt/os16.b/bin
30343                    else
30344                        echo "[$0] Cannot copy the application "
30345                        echo "[$0]   $EXEC inside the floppy "
30346                        echo "[$0]   disk image!"
30347                        break
30348                    fi
30349                fi
30350            fi
30351        done
30352        sync
30353        #
30354        # Altre applicazioni più importanti.
30355        #
30356        for d in ported/*
30357        do
30358            if [ -d "$d" ]
30359            then
30360                #
30361                #
30362                #
30363                OBJECTS=""
30364                BASENAME=`basename $d`
30365                EXEC="$d/$BASENAME"
30366                #
30367                #
30368                #
30369                if [ "$BASENAME" = "mix" ]
30370                then
30371                    #
30372                    # già fatto.
30373                    #
30374                    continue
30375                fi
30376                #
30377                #
30378                #
30379                for o in $d/*.o
30380                do
30381                    if    [ "$o" = "$d/crt0.o" ] \
```

```
30382                         || [ ! -e "$o" ]
30383                      then
30384                          true
30385                      else
30386                          OBJECTS="$OBJECTS $o"
30387                      fi
30388                  done
30389                  #
30390                  ld86 -i -d -s -o $EXEC $d/crt0.o $OBJECTS $OBJLIB
30391                  #
30392                  if [ -x "$d/$BASENAME" ]
30393                  then
30394                      if mount | grep /mnt/os16.b > /dev/null
30395                      then
30396                          mkdir /mnt/os16.b/bin/ 2> /dev/null
30397                          cp -f "$EXEC" /mnt/os16.b/bin
30398                      else
30399                          echo "[$0] Cannot copy the application "
30400                          echo "[$0]   $BASENAME inside the floppy "
30401                          echo "[$0]   disk image!"
30402                          break
30403                      fi
30404                  fi
30405              fi
30406          done
30407          sync
30408
30409      fi
30410  }
30411  #
30412  # Start.
30413  #
30414  if [ -d kernel ]   && \
30415     [ -d applic ]   && \
30416     [ -d lib  ]
30417  then
30418      OS16PATH=`pwd`
30419      main
30420  else
30421      echo "[$0] Running from a wrong directory!"
30422  fi
```

100.2 os16: «kernel/devices.h»

Si veda la sezione 99.1.

```
40001 #ifndef _KERNEL_DEVICES_H
40002 #define _KERNEL_DEVICES_H 1
40003
40004 #include <sys/os16.h>
40005 #include <sys/types.h>
40006 //----------------------------------------------------------------------
40007 #define DEV_READ                0
40008 #define DEV_WRITE               1
40009 ssize_t dev_io    (pid_t pid, dev_t device, int rw, off_t offset,
40010                    void *buffer, size_t size, int *eof);
40011 //----------------------------------------------------------------------
40012 // The following functions are used only by 'dev_io()'.
40013 //----------------------------------------------------------------------
40014 ssize_t dev_mem   (pid_t pid, dev_t device, int rw, off_t offset,
40015                    void *buffer, size_t size, int *eof);
40016 ssize_t dev_tty   (pid_t pid, dev_t device, int rw, off_t offset,
40017                    void *buffer, size_t size, int *eof);
40018 ssize_t dev_dsk   (pid_t pid, dev_t device, int rw, off_t offset,
40019                    void *buffer, size_t size, int *eof);
40020 ssize_t dev_kmem  (pid_t pid, dev_t device, int rw, off_t offset,
40021                    void *buffer, size_t size, int *eof);
40022 //----------------------------------------------------------------------
40023
40024 #endif
```

100.2.1 kernel/devices/dev_dsk.c

Si veda la sezione 99.1.2.

```
50001 #include <sys/os16.h>
50002 #include <kernel/devices.h>
50003 #include <sys/types.h>
50004 #include <errno.h>
50005 #include <kernel/memory.h>
50006 #include <kernel/ibm_i86.h>
50007 #include <kernel/proc.h>
50008 #include <string.h>
50009 #include <signal.h>
50010 #include <kernel/k_libc.h>
50011 #include <ctype.h>
50012 #include <kernel/tty.h>
50013 //----------------------------------------------------------------------
50014 ssize_t
50015 dev_dsk (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
50016          size_t size, int *eof)
```

```
50017 {
50018     ssize_t n;
50019     int     dev_minor = minor (device);
50020
50021     if (rw == DEV_READ)
50022       {
50023         n = dsk_read_bytes (dev_minor, offset, buffer, size);
50024       }
50025     else
50026       {
50027         n = dsk_write_bytes (dev_minor, offset, buffer, size);
50028       }
50029     return (n);
50030 }
```

100.2.2 kernel/devices/dev_io.c

«

Si veda la sezione 99.1.1.

```
60001 #include <sys/os16.h>
60002 #include <kernel/devices.h>
60003 #include <sys/types.h>
60004 #include <errno.h>
60005 #include <kernel/memory.h>
60006 #include <kernel/ibm_i86.h>
60007 #include <kernel/proc.h>
60008 #include <string.h>
60009 #include <signal.h>
60010 #include <kernel/k_libc.h>
60011 #include <ctype.h>
60012 #include <kernel/tty.h>
60013 //----------------------------------------------------------------------
60014 ssize_t
60015 dev_io (pid_t pid, dev_t device, int rw, off_t offset,
60016         void *buffer, size_t size, int *eof)
60017 {
60018     int dev_major = major (device);
60019     if (rw != DEV_READ && rw != DEV_WRITE)
60020       {
60021         errset (EIO);
60022         return (-1);
60023       }
60024     switch (dev_major)
60025       {
60026         case DEV_MEM_MAJOR:
60027             return (dev_mem (pid, device, rw, offset, buffer, size,
60028                                       eof));
60029         case DEV_TTY_MAJOR:
```

```
60030                 return (dev_tty (pid, device, rw, offset, buffer, size,
60031                                  eof));
60032             case DEV_CONSOLE_MAJOR:
60033                 return (dev_tty (pid, device, rw, offset, buffer, size,
60034                                  eof));
60035             case DEV_DSK_MAJOR:
60036                 return (dev_dsk (pid, device, rw, offset, buffer, size,
60037                                  eof));
60038             case DEV_KMEM_MAJOR:
60039                 return (dev_kmem (pid, device, rw, offset, buffer, size,
60040                                   eof));
60041             default:
60042                 errset (ENODEV);
60043                 return (-1);
60044         }
60045 }
```

100.2.3 kernel/devices/dev_kmem.c

«

Si veda la sezione 99.1.3.

```
70001 #include <sys/os16.h>
70002 #include <kernel/devices.h>
70003 #include <sys/types.h>
70004 #include <errno.h>
70005 #include <kernel/memory.h>
70006 #include <kernel/ibm_i86.h>
70007 #include <kernel/proc.h>
70008 #include <string.h>
70009 #include <signal.h>
70010 #include <kernel/k_libc.h>
70011 #include <ctype.h>
70012 #include <kernel/tty.h>
70013 //----------------------------------------------------------------------
70014 ssize_t
70015 dev_kmem (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
70016           size_t size, int *eof)
70017 {
70018     size_t    size_real;
70019     inode_t  *inode;
70020     sb_t     *sb;
70021     file_t   *file;
70022     void     *start;
70023     //
70024     // Only read is allowed.
70025     //
70026     if (rw != DEV_READ)
70027       {
```

```
70028         errset (EIO);                          // I/O error.
70029         return ((ssize_t) -1);
70030       }
70031     //
70032     // Only positive offset is allowed.
70033     //
70034     if (offset < 0)
70035       {
70036         errset (EIO);                          // I/O error.
70037         return ((ssize_t) -1);
70038       }
70039     //
70040     // Read is selected (and is the only access allowed).
70041     //
70042     switch (device)
70043       {
70044         case DEV_KMEM_PS:
70045             //
70046             // Verify if the selected slot can be read.
70047             //
70048             if (offset >= PROCESS_MAX)
70049               {
70050                 errset (EIO);                  // I/O error.
70051                 return ((ssize_t) -1);
70052               }
70053             //
70054             // Correct the size to be read.
70055             //
70056             if (sizeof (proc_t) < size)
70057               {
70058                 size = sizeof (proc_t);
70059               }
70060             //  //
70061             //  // Correct the size to be read.
70062             //  //
70063             //  size_real = ((sizeof (proc_t)) * (PROCESS_MAX - offset));
70064             //  if (size_real < size)
70065             //    {
70066             //      size = size_real;
70067             //    }
70068             //
70069             // Get the pointer to the selected slot.
70070             //
70071             start = proc_reference ((pid_t) offset);
70072             break;
70073         case DEV_KMEM_MMP:
70074             //
70075             // Correct the size to be read.
```

```
70076                //
70077                size_real = (MEM_MAX_BLOCKS/8);
70078                if (size_real < size)
70079                  {
70080                    size = size_real;
70081                  }
70082                //
70083                // Get the pointer to the map.
70084                //
70085                start = mb_reference ();
70086                break;
70087            case DEV_KMEM_SB:
70088                //
70089                // Get a reference to the super block table.
70090                //
70091                sb = sb_reference (0);
70092                //
70093                // Correct the size to be read.
70094                //
70095                if (sizeof (sb_t) < size)
70096                  {
70097                    size = sizeof (sb_t);
70098                  }
70099                //
70100                // Get the pointer to the selected super block slot.
70101                //
70102                start = &sb[offset];
70103                break;
70104            case DEV_KMEM_INODE:
70105                //
70106                // Get a reference to the inode table.
70107                //
70108                inode = inode_reference (0, 0);
70109                //
70110                // Correct the size to be read.
70111                //
70112                if (sizeof (inode_t) < size)
70113                  {
70114                    size = sizeof (inode_t);
70115                  }
70116                //
70117                // Get the pointer to the selected inode slot.
70118                //
70119                start = &inode[offset];
70120                break;
70121            case DEV_KMEM_FILE:
70122                //
70123                // Get a reference to the file table.
```

```
70124               //
70125               file = file_reference (0);
70126               //
70127               // Correct the size to be read.
70128               //
70129               if (sizeof (file_t) < size)
70130                 {
70131                   size = sizeof (file_t);
70132                 }
70133               //
70134               // Get the pointer to the selected inode slot.
70135               //
70136               start = &file[offset];
70137               break;
70138           default:
70139               errset (ENODEV);               // No such device.
70140               return ((ssize_t) -1);
70141       }
70142     //
70143     // At this point, data is ready to be copied to the buffer.
70144     //
70145     memcpy (buffer, start, size);
70146     //
70147     // Return size read.
70148     //
70149     return (size);
70150 }
```

100.2.4 kernel/devices/dev_mem.c

«

Si veda la sezione 99.1.4.

```
80001 #include <sys/os16.h>
80002 #include <kernel/devices.h>
80003 #include <sys/types.h>
80004 #include <errno.h>
80005 #include <kernel/memory.h>
80006 #include <kernel/ibm_i86.h>
80007 #include <kernel/proc.h>
80008 #include <string.h>
80009 #include <signal.h>
80010 #include <kernel/k_libc.h>
80011 #include <ctype.h>
80012 #include <kernel/tty.h>
80013 //----------------------------------------------------------------------
80014 ssize_t
80015 dev_mem (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
80016           size_t size, int *eof)
```

```
80017 {
80018     uint8_t  *buffer08 = (uint8_t *) buffer;
80019     uint16_t *buffer16 = (uint16_t *) buffer;
80020     ssize_t n;
80021 
80022     if      (device == DEV_MEM)                         // DEV_MEM
80023       {
80024         if (rw == DEV_READ)
80025           {
80026             n = mem_read ((addr_t) offset, buffer, size);
80027           }
80028         else
80029           {
80030             n = mem_write ((addr_t) offset, buffer, size);
80031           }
80032       }
80033     else if (device == DEV_NULL)                        // DEV_NULL
80034       {
80035         n = 0;
80036       }
80037     else if (device == DEV_ZERO)                        // DEV_ZERO
80038       {
80039         if (rw == DEV_READ)
80040           {
80041             for (n = 0; n < size; n++)
80042               {
80043                 buffer08[n] = 0;
80044               }
80045           }
80046         else
80047           {
80048             n = 0;
80049           }
80050       }
80051     else if (device == DEV_PORT)                        // DEV_PORT
80052       {
80053         if (rw == DEV_READ)
80054           {
80055             if (size == 1)
80056               {
80057                 buffer08[0] = in_8 (offset);
80058                 n = 1;
80059               }
80060             else if (size == 2)
80061               {
80062                 buffer16[0] = in_16 (offset);
80063                 n = 2;
80064               }
```

```
80065                 else
80066                   {
80067                     n = 0;
80068                   }
80069               }
80070             else
80071               {
80072                 if (size == 1)
80073                   {
80074                     out_8 (offset, buffer08[0]);
80075                   }
80076                 else if (size == 2)
80077                   {
80078                     out_16 (offset, buffer16[0]);
80079                     n = 2;
80080                   }
80081                 else
80082                   {
80083                     n = 0;
80084                   }
80085               }
80086           }
80087         else
80088           {
80089             errset (ENODEV);
80090             return (-1);
80091           }
80092         return (n);
80093 }
```

100.2.5 kernel/devices/dev_tty.c

Si veda la sezione 99.1.5.

```
90001 #include <sys/os16.h>
90002 #include <kernel/devices.h>
90003 #include <sys/types.h>
90004 #include <errno.h>
90005 #include <kernel/memory.h>
90006 #include <kernel/ibm_i86.h>
90007 #include <kernel/proc.h>
90008 #include <string.h>
90009 #include <signal.h>
90010 #include <kernel/k_libc.h>
90011 #include <ctype.h>
90012 #include <kernel/tty.h>
90013 //----------------------------------------------------------------------
90014 ssize_t
```

```
90015 dev_tty (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
90016           size_t size, int *eof)
90017 {
90018     uint8_t *buffer08 = (uint8_t *) buffer;
90019     ssize_t  n;
90020     proc_t  *ps;
90021     //
90022     // Get process. Variable 'ps' will be 'NULL' if the process ID is
90023     // not valid.
90024     //
90025     ps = proc_reference (pid);
90026     //
90027     // Convert 'DEV_TTY' with the controlling terminal for the process.
90028     //
90029     if (device == DEV_TTY)
90030       {
90031         device = ps->device_tty;
90032         //
90033         // As a last resort, use the generic 'DEV_CONSOLE'.
90034         //
90035         if (device == 0 || device == DEV_TTY)
90036           {
90037             device = DEV_CONSOLE;
90038           }
90039       }
90040     //
90041     // Convert 'DEV_CONSOLE' to the currently active console.
90042     //
90043     if (device == DEV_CONSOLE)
90044       {
90045         device = tty_console ((dev_t) 0);
90046         //
90047         // As a last resort, use the first console: 'DEV_CONSOLE0'.
90048         //
90049         if (device == 0 || device == DEV_TTY)
90050           {
90051             device = DEV_CONSOLE0;
90052           }
90053       }
90054     //
90055     // Read or write.
90056     //
90057     if (rw == DEV_READ)
90058       {
90059         for (n = 0; n < size; n++)
90060           {
90061             buffer08[n] = tty_read (device);
90062             if (buffer08[n] == 0)
```

```
90063                 {
90064                   //
90065                   // If the pid is not the kernel, should put the process
90066                   // to sleep, waiting for the key.
90067                   //
90068                   if (pid == 0 || ps == NULL)
90069                     {
90070                       //
90071                       // For the kernel there is no sleep and for an
90072                       // unidentified process, either.
90073                       //
90074                       break;
90075                     }
90076                   //
90077                   // Put the process to sleep.
90078                   //
90079                   ps->status         = PROC_SLEEPING;
90080                   ps->ret            = 0;
90081                   ps->wakeup_events  = WAKEUP_EVENT_TTY;
90082                   ps->wakeup_signal  = 0;
90083                   ps->wakeup_timer   = 0;
90084                   //
90085                   break;
90086                 }
90087               //
90088               // Check for control characters.
90089               //
90090               if (buffer08[n] == 0x04)    // EOT
90091                 {
90092                   //
90093                   // Return EOF.
90094                   //
90095                   *eof = 1;
90096                   break;
90097                 }
90098               //
90099               // At this point, show the character on screen, even if it
90100               // is not nice. It is necessary to show something, because
90101               // the tty handling is very poor and the library for line
90102               // input, calculate cursor position based on the characters
90103               // received.
90104               //
90105               tty_write (device, (int) buffer08[n]);
90106             }
90107         }
90108       else
90109         {
90110           for (n = 0; n < size; n++)
```

```
90111              {
90112                tty_write (device, (int) buffer08[n]);
90113              }
90114          }
90115        return (n);
90116  }
```

100.3 os16: «kernel/diag.h»

Si veda la sezione 99.2.

```
100001 #ifndef _KERNEL_DIAG_H
100002 #define _KERNEL_DIAG_H 1
100003
100004 #include <stdint.h>
100005 #include <kernel/fs.h>
100006 #include <sys/types.h>
100007 #include <kernel/proc.h>
100008
100009 //----------------------------------------------------------------------
100010 uint8_t  reverse_8_bit  (uint8_t  source);
100011 uint16_t reverse_16_bit (uint16_t source);
100012 uint32_t reverse_32_bit (uint32_t source);
100013
100014 #define reverse_char(s)      ((char)     reverse_8_bit  ((uint8_t)  s))
100015 #define reverse_short(s)     ((short)    reverse_16_bit ((uint16_t) s))
100016 #define reverse_int(s)       ((int)      reverse_16_bit ((uint16_t) s))
100017 #define reverse_long(s)      ((long)     reverse_32_bit ((uint32_t) s))
100018 #define reverse_long_int(s)  ((long int) reverse_32_bit ((uint32_t) s))
100019 //----------------------------------------------------------------------
100020 void print_hex_8  (void *data, size_t elements);
100021 void print_hex_16 (void *data, size_t elements);
100022 void print_hex_32 (void *data, size_t elements);
100023
100024 #define print_hex_char(d, e)      (print_hex_8  (d, e))
100025 #define print_hex_short(d, e)     (print_hex_16 (d, e))
100026 #define print_hex_int(d, e)       (print_hex_16 (d, e))
100027 #define print_hex_long(d, e)      (print_hex_32 (d, e))
100028 #define print_hex_long_int(d, e)  (print_hex_32 (d, e))
100029 //----------------------------------------------------------------------
100030 void print_hex_8_reverse  (void *data, size_t elements);
100031 void print_hex_16_reverse (void *data, size_t elements);
100032 void print_hex_32_reverse (void *data, size_t elements);
100033
100034 #define print_hex_char_reverse(d, e)      (print_hex_8_reverse  (d, e))
100035 #define print_hex_short_reverse(d, e)     (print_hex_16_reverse (d, e))
100036 #define print_hex_int_reverse(d, e)       (print_hex_16_reverse (d, e))
100037 #define print_hex_long_reverse(d, e)      (print_hex_32_reverse (d, e))
```

```
100038 #define print_hex_long_int_reverse(d, e)  (print_hex_32_reverse (d, e))
100039 //----------------------------------------------------------------------
100040 void print_segments (void);
100041 void print_kmem     (void);
100042 //----------------------------------------------------------------------
100043 void print_mb_map     (void);
100044 void print_memory_map (void);
100045 //----------------------------------------------------------------------
100046 void print_superblock        (sb_t *sb);
100047 void print_inode             (inode_t *inode);
100048 void print_inode_map         (sb_t *sb, uint16_t *bitmap);
100049 void print_zone_map          (sb_t *sb, uint16_t *bitmap);
100050 void print_inode_head        (void);
100051 void print_inode_list        (void);
100052 void print_inode_zones_head (void);
100053 void print_inode_zones       (inode_t *inode);
100054 void print_inode_zones_list (void);
100055 //----------------------------------------------------------------------
100056 void print_proc_head (void);
100057 void print_proc_pid  (proc_t *ps, pid_t pid);
100058 void print_proc_list (void);
100059 //----------------------------------------------------------------------
100060 void print_file_head (void);
100061 void print_file_num  (int num);
100062 void print_file_list (void);
100063 //----------------------------------------------------------------------
100064 void print_fd_head (void);
100065 void print_fd      (fd_t *fd);
100066 void print_fd_list (pid_t pid);
100067 //----------------------------------------------------------------------
100068 void print_time      (void);
100069 //----------------------------------------------------------------------
100070 
100071 #endif
```

100.3.1 kernel/diag/print_fd.c

«

Si veda la sezione 99.2.

```
110001 #include <sys/os16.h>
110002 #include <kernel/diag.h>
110003 #include <kernel/k_libc.h>
110004 #include <fcntl.h>
110005 //----------------------------------------------------------------------
110006 void
110007 print_fd (fd_t *fd)
110008 {
110009     k_printf ("%04x %6li %3i %c/%c %05o %5i %3i %5li %4i %04x %3i",
```

```
110010            (unsigned int)      fd->fl_flags,
110011            (unsigned long int) fd->file->offset,
110012            (unsigned int)      fd->file->references,
110013            (fd->file->oflags & O_RDONLY ? 'r' : ' '),
110014            (fd->file->oflags & O_WRONLY ? 'w' : ' '),
110015            (unsigned int)      fd->file->inode->mode,
110016            (unsigned int)      fd->file->inode->uid,
110017            (unsigned int)      fd->file->inode->gid,
110018            (unsigned long int) fd->file->inode->size,
110019            (unsigned int)      fd->file->inode->links,
110020            (unsigned int)      fd->file->inode->sb->device,
110021            (unsigned int)      fd->file->inode->ino);
110022        k_printf ("\n");
110023    }
```

100.3.2 kernel/diag/print_fd_head.c

Si veda la sezione 99.2.

```
120001 #include <sys/os16.h>
120002 #include <kernel/diag.h>
120003 #include <kernel/k_libc.h>
120004 //----------------------------------------------------------------------
120005 void
120006 print_fd_head (void)
120007 {
120008
120009     k_printf ("n. stat offset ref flg  mode   uid gid  size lnks  ");
120010     k_printf ("dev ino\n");
120011 }
```

100.3.3 kernel/diag/print_fd_list.c

Si veda la sezione 99.2.

```
130001 #include <sys/os16.h>
130002 #include <kernel/diag.h>
130003 #include <kernel/k_libc.h>
130004 //----------------------------------------------------------------------
130005 void
130006 print_fd_list (pid_t pid)
130007 {
130008     int      fdn = 0;
130009     fd_t *fd;
130010     fd = fd_reference (pid, &fdn);
130011     print_fd_head ();
130012     for (fdn = 0; fdn < OPEN_MAX; fdn++)
```

```
130013         {
130014           if (fd[fdn].file != NULL)
130015             {
130016               k_printf ("%2i ", fdn);
130017               print_fd (fd);
130018             }
130019         }
130020 }
```

100.3.4 kernel/diag/print_file_head.c

«

Si veda la sezione 99.2.

```
140001 #include <sys/os16.h>
140002 #include <kernel/diag.h>
140003 #include <kernel/k_libc.h>
140004 //----------------------------------------------------------------------
140005 void
140006 print_file_head (void)
140007 {
140008 
140009     k_printf ("n. ref flg  mode uid  size lnks  dev ino\n");
140010 }
```

100.3.5 kernel/diag/print_file_list.c

«

Si veda la sezione 99.2.

```
150001 #include <sys/os16.h>
150002 #include <kernel/diag.h>
150003 #include <kernel/k_libc.h>
150004 //----------------------------------------------------------------------
150005 void
150006 print_file_list (void)
150007 {
150008     int     fno;
150009     file_t *file = file_reference (0);
150010     //
150011     print_file_head ();
150012     //
150013     for (fno = 0; fno < FILE_MAX_SLOTS; fno++)
150014       {
150015         if (file[fno].references > 0)
150016           {
150017             print_file_num (fno);
150018           }
150019       }
```

```
150020 }
```

100.3.6 kernel/diag/print_file_num.c

Si veda la sezione 99.2.

```
160001 #include <sys/os16.h>
160002 #include <kernel/diag.h>
160003 #include <kernel/k_libc.h>
160004 #include <fcntl.h>
160005 //----------------------------------------------------------------------
160006 void
160007 print_file_num (int fno)
160008 {
160009     file_t *file = file_reference (fno);
160010 
160011     k_printf ("%2i %3i %c/%c %05o %3i %5li %4i %04x %3i",
160012                  (unsigned int)      fno,
160013                  (unsigned int)      file->references,
160014                  (file->oflags & O_RDONLY ? 'r' : ' '),
160015                  (file->oflags & O_WRONLY ? 'w' : ' '),
160016                  (unsigned int)      file->inode->mode,
160017                  (unsigned int)      file->inode->uid,
160018                  (unsigned long int) file->inode->size,
160019                  (unsigned int)      file->inode->links,
160020                  (unsigned int)      file->inode->sb->device,
160021                  (unsigned int)      file->inode->ino);
160022     k_printf ("\n");
160023 }
```

100.3.7 kernel/diag/print_hex_16.c

Si veda la sezione 99.2.

```
170001 #include <sys/os16.h>
170002 #include <kernel/diag.h>
170003 #include <kernel/k_libc.h>
170004 #include <inttypes.h>
170005 #include <stdio.h>
170006 //----------------------------------------------------------------------
170007 void
170008 print_hex_16 (void *data, size_t elements)
170009 {
170010     uint16_t *element = (uint16_t *) data;
170011     int i;
170012     for (i = 0; i < elements; i++)
170013       {
```

```
170014          k_printf ("%04" PRIx16, (uint16_t) element[i]);
170015        }
170016  }
```

100.3.8 kernel/diag/print_hex_16_reverse.c

Si veda la sezione 99.2.

```
180001  #include <sys/os16.h>
180002  #include <kernel/diag.h>
180003  #include <kernel/k_libc.h>
180004  #include <inttypes.h>
180005  #include <stdio.h>
180006  //----------------------------------------------------------------------
180007  void
180008  print_hex_16_reverse (void *data, size_t elements)
180009  {
180010      uint16_t *element = (uint16_t *) data;
180011      int i;
180012      for (i = 0; i < elements; i++)
180013        {
180014          k_printf ("%04" PRIx16, reverse_16_bit (element[i]));
180015        }
180016  }
```

100.3.9 kernel/diag/print_hex_32.c

Si veda la sezione 99.2.

```
190001  #include <sys/os16.h>
190002  #include <kernel/diag.h>
190003  #include <kernel/k_libc.h>
190004  #include <inttypes.h>
190005  #include <stdio.h>
190006  //----------------------------------------------------------------------
190007  void
190008  print_hex_32 (void *data, size_t elements)
190009  {
190010      uint32_t *element = (uint32_t *) data;
190011      int i;
190012      for (i = 0; i < elements; i++)
190013        {
190014          k_printf ("%08" PRIx32, (uint32_t) element[i]);
190015        }
190016  }
```

100.3.10 kernel/diag/print_hex_32_reverse.c

Si veda la sezione 99.2.

```
200001 #include <sys/os16.h>
200002 #include <kernel/diag.h>
200003 #include <kernel/k_libc.h>
200004 #include <inttypes.h>
200005 #include <stdio.h>
200006 //----------------------------------------------------------------------
200007 void
200008 print_hex_32_reverse (void *data, size_t elements)
200009 {
200010     uint32_t *element = (uint32_t *) data;
200011     int i;
200012     for (i = 0; i < elements; i++)
200013       {
200014         k_printf ("%08" PRIx32, reverse_32_bit (element[i]));
200015       }
200016 }
```

100.3.11 kernel/diag/print_hex_8.c

Si veda la sezione 99.2.

```
210001 #include <sys/os16.h>
210002 #include <kernel/diag.h>
210003 #include <kernel/k_libc.h>
210004 #include <inttypes.h>
210005 #include <stdio.h>
210006 //----------------------------------------------------------------------
210007 void
210008 print_hex_8 (void *data, size_t elements)
210009 {
210010     uint8_t *element = (uint8_t *) data;
210011     int i;
210012     for (i = 0; i < elements; i++)
210013       {
210014         k_printf ("%02" PRIx8, (uint16_t) element[i]);
210015       }
210016 }
```

100.3.12 kernel/diag/print_hex_8_reverse.c

«

Si veda la sezione 99.2.

```
220001 #include <sys/os16.h>
220002 #include <kernel/diag.h>
220003 #include <kernel/k_libc.h>
220004 #include <inttypes.h>
220005 #include <stdio.h>
220006 //----------------------------------------------------------------------
220007 void
220008 print_hex_8_reverse (void *data, size_t elements)
220009 {
220010     uint8_t *element = (uint8_t *) data;
220011     int i;
220012     for (i = 0; i < elements; i++)
220013       {
220014         k_printf ("%02" PRIx8, reverse_8_bit (element[i]));
220015       }
220016 }
```

100.3.13 kernel/diag/print_inode.c

«

Si veda la sezione 99.2.

```
230001 #include <sys/os16.h>
230002 #include <kernel/diag.h>
230003 #include <kernel/k_libc.h>
230004 //----------------------------------------------------------------------
230005 void
230006 print_inode (inode_t *inode)
230007 {
230008     unsigned long int seconds;
230009     unsigned long int seconds_d;
230010     unsigned long int seconds_h;
230011     unsigned int      d;
230012     unsigned int      h;
230013     unsigned int      m;
230014     unsigned int      s;
230015     dev_t             device_attached = 0;
230016     //
230017     if (inode == NULL)
230018       {
230019         return;
230020       }
230021     //
230022     seconds    = inode->time;
230023     d          = seconds / 86400L;              // 24 * 60 * 60
230024     seconds_d  = d;
```

```
230025      seconds_d *= 86400;
230026      seconds   -= seconds_d;
230027      h          = seconds / 3840;                // 60 * 60
230028      seconds_h  = h;
230029      seconds_h *= 3840;
230030      seconds   -= seconds_h;
230031      m          = seconds / 60;
230032      s          = seconds % 60;
230033      //
230034      if (inode->sb_attached != NULL)
230035        {
230036          device_attached = inode->sb_attached->device;
230037        }
230038      //
230039      k_printf ("%04x %4i %3i %c %4x %06o %4i %3i %7li ",
230040                (unsigned int) inode->sb->device,
230041                (unsigned int) inode->ino,
230042                (unsigned int) inode->references,
230043                (inode->changed ? '!' : ' '),
230044                (unsigned int) device_attached,
230045                (unsigned int) inode->mode,
230046                (unsigned int) inode->uid,
230047                (unsigned int) inode->gid,
230048                (unsigned long int) inode->size);
230049
230050      k_printf ("%5i %2i:%02i:%02i %3i\n",
230051                d, h, m, s,
230052                (unsigned int) inode->links);
230053
230054  }
```

100.3.14 kernel/diag/print_inode_head.c

Si veda la sezione 99.2.

```
240001  #include <sys/os16.h>
240002  #include <kernel/diag.h>
240003  #include <kernel/k_libc.h>
240004  //----------------------------------------------------------------------
240005  void
240006  print_inode_head (void)
240007  {
240008      k_printf (" dev  ino ref c mntd   mode  uid gid     ");
240009      k_printf ("size  date     time lnk \n");
240010  }
```

100.3.15 kernel/diag/print_inode_list.c

«

Si veda la sezione 99.2.

```
250001 #include <sys/os16.h>
250002 #include <kernel/diag.h>
250003 #include <kernel/k_libc.h>
250004 //----------------------------------------------------------------------
250005 void
250006 print_inode_list (void)
250007 {
250008     ino_t ino;
250009     inode_t *inode = inode_reference (0, 0);
250010     print_inode_head ();
250011     for (ino = 0; ino < INODE_MAX_SLOTS; ino++)
250012       {
250013         if (inode[ino].references > 0)
250014           {
250015             print_inode (&inode[ino]);
250016           }
250017       }
250018 }
```

100.3.16 kernel/diag/print_inode_map.c

«

Si veda la sezione 99.2.

```
260001 #include <sys/os16.h>
260002 #include <kernel/diag.h>
260003 #include <kernel/k_libc.h>
260004 //----------------------------------------------------------------------
260005 void
260006 print_inode_map (sb_t *sb, uint16_t *bitmap)
260007 {
260008     size_t size;
260009     if (sb->inodes % 16)
260010       {
260011         size = sb->inodes/16 + 1;
260012       }
260013     else
260014       {
260015         size = sb->inodes/16;
260016       }
260017     print_hex_16_reverse (bitmap, size);
260018 }
```

100.3.17 kernel/diag/print_inode_zone_list.c

Si veda la sezione 99.2.

```
270001 #include <sys/os16.h>
270002 #include <kernel/diag.h>
270003 #include <kernel/k_libc.h>
270004 //----------------------------------------------------------------------
270005 void
270006 print_inode_zones_list (void)
270007 {
270008     ino_t ino;
270009     inode_t *inode = inode_reference (0, 0);
270010     print_inode_zones_head ();
270011     for (ino = 0; ino < INODE_MAX_SLOTS; ino++)
270012       {
270013         if (inode[ino].references > 0)
270014           {
270015             print_inode_zones (&inode[ino]);
270016           }
270017       }
270018 }
```

100.3.18 kernel/diag/print_inode_zones.c

Si veda la sezione 99.2.

```
280001 #include <sys/os16.h>
280002 #include <kernel/diag.h>
280003 #include <kernel/k_libc.h>
280004 //----------------------------------------------------------------------
280005 void
280006 print_inode_zones (inode_t *inode)
280007 {
280008     int i;
280009     //
280010     if (inode == NULL)
280011       {
280012         return;
280013       }
280014     //
280015     k_printf ("%04x %4i ",
280016               (unsigned int) inode->sb->device,
280017               (unsigned int) inode->ino);
280018
280019     for (i = 0; i < 7; i++)
280020       {
280021         if (inode->direct[i] != 0)
280022           {
```

```
280023                k_printf ("%04x ", (unsigned int) inode->direct[i]);
280024              }
280025            else
280026              {
280027                k_printf ("     ");
280028              }
280029          }
280030        if (inode->indirect1 != 0)
280031          {
280032            k_printf ("%04x ",      (unsigned int) inode->indirect1);
280033          }
280034        else
280035          {
280036            k_printf ("     ");
280037          }
280038        if (inode->indirect2 != 0)
280039          {
280040            k_printf ("%04x",       (unsigned int) inode->indirect2);
280041          }
280042        else
280043          {
280044            k_printf ("    ");
280045          }
280046        k_printf ("\n");
280047    }
```

100.3.19 kernel/diag/print_inode_zones_head.c

«

Si veda la sezione 99.2.

```
290001  #include <sys/os16.h>
290002  #include <kernel/diag.h>
290003  #include <kernel/k_libc.h>
290004  //----------------------------------------------------------------------
290005  void
290006  print_inode_zones_head (void)
290007  {
290008      k_printf (" dev  ino zn_0 zn_1 zn_2 zn_3 zn_4 zn_5 zn_6 ");
290009      k_printf ("ind1 ind2\n");
290010  }
```

100.3.20 kernel/diag/print_kmem.c

Si veda la sezione 99.2.

```
300001 #include <sys/os16.h>
300002 #include <kernel/diag.h>
300003 #include <kernel/k_libc.h>
300004 //----------------------------------------------------------------------
300005 extern uint16_t _ksp;
300006 extern uint16_t _etext;
300007 extern uint16_t _edata;
300008 extern uint16_t _end;
300009 //----------------------------------------------------------------------
300010 void
300011 print_kmem (void)
300012 {
300013     k_printf ("etext=%04x edata=%04x ebss=%04x ksp=%04x",
300014               (unsigned int) &_etext,
300015               (unsigned int) &_edata, (unsigned int) &_end, _ksp);
300016 }
```

100.3.21 kernel/diag/print_mb_map.c

Si veda la sezione 99.2.

```
310001 #include <sys/os16.h>
310002 #include <kernel/diag.h>
310003 #include <kernel/k_libc.h>
310004 //----------------------------------------------------------------------
310005 void
310006 print_mb_map (void)
310007 {
310008     uint16_t    *mb = mb_reference ();
310009     unsigned int i;
310010     for (i = 0; i < (MEM_MAX_BLOCKS / 16); i++)
310011       {
310012         k_printf ("%04x", mb[i]);
310013       }
310014 }
```

100.3.22 kernel/diag/print_memory_map.c

Si veda la sezione 99.2.

```
320001 #include <sys/os16.h>
320002 #include <kernel/diag.h>
320003 #include <kernel/k_libc.h>
320004 //----------------------------------------------------------------------
```

```
320005 void
320006 print_memory_map (void)
320007 {
320008     uint16_t    *mem_block[MEM_BLOCK_SIZE/2];
320009     uint16_t     block_rank;
320010     unsigned int b;
320011     unsigned int m;
320012     unsigned int i;
320013     addr_t       start;
320014 
320015     start = 0;
320016     dev_io ((pid_t) -1, DEV_MEM, DEV_READ, start, mem_block,
320017             MEM_BLOCK_SIZE, NULL);
320018 
320019     for (m = 0; m < MEM_MAX_BLOCKS; m++)
320020       {
320021         i = m % 16;
320022         if (i == 0)
320023           {
320024             block_rank = 0;
320025           }
320026         //
320027         for (b = 0; b < (MEM_BLOCK_SIZE / 2); b++)
320028           {
320029             if (mem_block[b])
320030               {
320031                 block_rank |= (0x8000 >> i);
320032                 break;
320033               }
320034           }
320035         //
320036         if (i == 15)
320037           {
320038             k_printf ("%04x", block_rank);
320039           }
320040         //
320041         start += MEM_BLOCK_SIZE;
320042         dev_io ((pid_t) -1, DEV_MEM, DEV_READ, start, mem_block,
320043                 MEM_BLOCK_SIZE, NULL);
320044       }
320045 }
```

100.3.23 kernel/diag/print_proc_head.c

Si veda la sezione 99.2.

```
330001 #include <sys/os16.h>
330002 #include <kernel/diag.h>
330003 #include <kernel/k_libc.h>
330004 //----------------------------------------------------------------------
330005 void
330006 print_proc_head (void)
330007 {
330008     k_printf (
330009 "pp  p pg                                                            \n"
330010 "id id rp  tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name\n"
330011            );
330012 }
```

100.3.24 kernel/diag/print_proc_list.c

Si veda la sezione 99.2.

```
340001 #include <sys/os16.h>
340002 #include <kernel/diag.h>
340003 #include <kernel/k_libc.h>
340004 //----------------------------------------------------------------------
340005 void
340006 print_proc_list (void)
340007 {
340008     pid_t   pid;
340009     proc_t *ps;
340010     //
340011     print_proc_head ();
340012     //
340013     for (pid = 0; pid < PROCESS_MAX; pid++)
340014       {
340015         ps = proc_reference (pid);
340016         if (ps != NULL && ps->status > 0)
340017           {
340018             print_proc_pid (ps, pid);
340019           }
340020       }
340021 }
```

100.3.25 kernel/diag/print_proc_pid.c

«

Si veda la sezione 99.2.

```
350001 #include <sys/os16.h>
350002 #include <kernel/diag.h>
350003 #include <kernel/k_libc.h>
350004 //----------------------------------------------------------------------
350005 void
350006 print_proc_pid (proc_t *ps, pid_t pid)
350007 {
350008     char  stat;
350009     switch (ps->status)
350010       {
350011         case PROC_EMPTY    : stat = '-'; break;
350012         case PROC_CREATED  : stat = 'c'; break;
350013         case PROC_READY    : stat = 'r'; break;
350014         case PROC_RUNNING  : stat = 'R'; break;
350015         case PROC_SLEEPING: stat = 's'; break;
350016         case PROC_ZOMBIE   : stat = 'z'; break;
350017         default            : stat = '?'; break;
350018       }
350019
350020     k_printf ("%2i %2i %2i %04x %4i %4i %4i %02i.%02i %c %05lx %04x ",
350021               (unsigned int) ps->ppid,
350022               (unsigned int) pid,
350023               (unsigned int) ps->pgrp,
350024               (unsigned int) ps->device_tty,
350025               (unsigned int) ps->uid,
350026               (unsigned int) ps->euid,
350027               (unsigned int) ps->suid,
350028               (unsigned int) ((ps->usage / CLOCKS_PER_SEC) / 60),
350029               (unsigned int) ((ps->usage / CLOCKS_PER_SEC) % 60),
350030               stat,
350031               (unsigned long int) ps->address_i,
350032               (unsigned int) ps->size_i);
350033
350034     k_printf ("%05lx %04x %04x %s",
350035               (unsigned long int) ps->address_d,
350036               (unsigned int) ps->size_d,
350037               (unsigned int) ps->sp,
350038                              ps->name);
350039
350040     k_printf ("\n");
350041 }
```

100.3.26 kernel/diag/print_segments.c

Si veda la sezione 99.2.

```
360001 #include <sys/os16.h>
360002 #include <kernel/diag.h>
360003 #include <kernel/k_libc.h>
360004 //----------------------------------------------------------------------
360005 void
360006 print_segments (void)
360007 {
360008     k_printf ("CS=%04x DS=%04x SS=%04x ES=%04x BP=%04x SP=%04x ",
360009               cs (), ds (), ss (), es (), bp (), sp ());
360010     k_printf ("heap_min=%04x", heap_min ());
360011 }
```

100.3.27 kernel/diag/print_superblock.c

Si veda la sezione 99.2.

```
370001 #include <sys/os16.h>
370002 #include <kernel/diag.h>
370003 #include <kernel/k_libc.h>
370004 //----------------------------------------------------------------------
370005 void
370006 print_superblock (sb_t *sb)
370007 {
370008     k_printf ("Inodes:           %i\n", sb->inodes);
370009     k_printf ("Blocks:           %i\n", sb->zones);
370010     k_printf ("First data zone:  %i\n", sb->first_data_zone);
370011     k_printf ("Zone size:        %i\n", (1024 << sb->log2_size_zone));
370012     k_printf ("Max file size:    %li\n", sb->max_file_size);
370013     k_printf ("Inode map blocks: %i\n", sb->map_inode_blocks);
370014     k_printf ("Zone map blocks:  %i\n", sb->map_zone_blocks);
370015 }
```

100.3.28 kernel/diag/print_time.c

Si veda la sezione 99.2.

```
380001 #include <sys/os16.h>
380002 #include <kernel/diag.h>
380003 #include <kernel/k_libc.h>
380004 //----------------------------------------------------------------------
380005 void
380006 print_time (void)
380007 {
380008     unsigned long int ticks   = k_clock ();
```

```
380009     unsigned long int seconds = k_time (NULL);
380010     unsigned int      h       = seconds / 60 / 60;
380011     unsigned int      m       = seconds / 60 - h * 60;
380012     unsigned int      s       = seconds - m * 60 - h * 60 * 60;
380013     k_printf ("clock=%08lx, time elapsed=%02u:%02u:%02u",
380014               ticks, h, m, s);
380015 }
```

100.3.29 kernel/diag/print_zone_map.c

«

Si veda la sezione 99.2.

```
390001 #include <sys/os16.h>
390002 #include <kernel/diag.h>
390003 #include <kernel/k_libc.h>
390004 //----------------------------------------------------------------------
390005 void
390006 print_zone_map (sb_t *sb, uint16_t *bitmap)
390007 {
390008     size_t       size;
390009     unsigned int data_zones = sb->zones - sb->first_data_zone;
390010     if (data_zones % 16)
390011       {
390012         size = data_zones/16 + 1;
390013       }
390014     else
390015       {
390016         size = data_zones/16;
390017       }
390018     print_hex_16_reverse (bitmap, size);
390019 }
```

100.3.30 kernel/diag/reverse_16_bit.c

«

Si veda la sezione 99.2.

```
400001 #include <sys/os16.h>
400002 #include <kernel/diag.h>
400003 #include <kernel/k_libc.h>
400004 #include <inttypes.h>
400005 //----------------------------------------------------------------------
400006 uint16_t
400007 reverse_16_bit (uint16_t source)
400008 {
400009     uint16_t destination = 0;
400010     uint16_t mask_src;
400011     uint16_t mask_dst;
```

```
400012      int       i;
400013      for (i = 0; i < 16; i++)
400014        {
400015          mask_src = 0x0001 << i;
400016          mask_dst = 0x8000 >> i;
400017          if (source & mask_src)
400018            {
400019              destination |= mask_dst;
400020            }
400021        }
400022      return (destination);
400023  }
```

100.3.31 kernel/diag/reverse_32_bit.c

Si veda la sezione 99.2.

```
410001  #include <sys/os16.h>
410002  #include <kernel/diag.h>
410003  #include <kernel/k_libc.h>
410004  //----------------------------------------------------------------------
410005  uint32_t
410006  reverse_32_bit (uint32_t source)
410007  {
410008      uint32_t destination = 0;
410009      uint32_t mask_src;
410010      uint32_t mask_dst;
410011      int       i;
410012      for (i = 0; i < 32; i++)
410013        {
410014          mask_src = 0x00000001 << i;
410015          mask_dst = 0x80000000 >> i;
410016          if (source & mask_src)
410017            {
410018              destination |= mask_dst;
410019            }
410020        }
410021      return (destination);
410022  }
```

100.3.32 kernel/diag/reverse_8_bit.c

Si veda la sezione 99.2.

```
420001  #include <sys/os16.h>
420002  #include <kernel/diag.h>
420003  #include <kernel/k_libc.h>
```

```
420004 #include <inttypes.h>
420005 //----------------------------------------------------------------------
420006 uint8_t
420007 reverse_8_bit (uint8_t source)
420008 {
420009     uint8_t destination = 0;
420010     uint8_t mask_src;
420011     uint8_t mask_dst;
420012     int      i;
420013     for (i = 0; i < 8; i++)
420014       {
420015         mask_src = 0x01 << i;
420016         mask_dst = 0x80 >> i;
420017         if (source & mask_src)
420018           {
420019             destination |= mask_dst;
420020           }
420021       }
420022     return (destination);
420023 }
```

100.4 os16: «kernel/fs.h»

«

Si veda la sezione 99.3.

```
430001 #ifndef _KERNEL_FS_H
430002 #define _KERNEL_FS_H 1
430003
430004 #include <stdint.h>
430005 #include <sys/types.h>
430006 #include <kernel/memory.h>
430007 #include <sys/os16.h>
430008 #include <sys/stat.h>
430009 #include <stdint.h>
430010 #include <const.h>
430011 #include <stdio.h>
430012 #include <limits.h>
430013
430014 //----------------------------------------------------------------------
430015 #define SB_MAX_INODE_BLOCKS          1  // 8192 inodes max.
430016 #define SB_MAX_ZONE_BLOCKS           1  // 8192 data-zones max.
430017 #define SB_BLOCK_SIZE             1024  // Fixed for Minix file system.
430018 #define SB_MAX_ZONE_SIZE          2048  // log2 max is 1.
430019 #define SB_MAP_INODE_SIZE         (SB_MAX_INODE_BLOCKS*512) // [1]
430020 #define SB_MAP_ZONE_SIZE          (SB_MAX_ZONE_BLOCKS*512)  // [1]
430021 //
430022 // [1] blocks * (1024 * 8 / 16) = number of bits, divided 16.
430023 //
```

```
430024 //----------------------------------------------------------------------
430025 #define INODE_MAX_INDIRECT_ZONES  (SB_MAX_ZONE_SIZE/2)      // [2]
430026
430027 #define INODE_MAX_REFERENCES      0xFF
430028 //
430029 // [2] number of zone pointers contained inside a zone, used
430030 //     as an indirect inode list (a pointer = 16 bits = 2 bytes).
430031 //
430032 //----------------------------------------------------------------------
430033 typedef uint16_t        zno_t;          // Zone number.
430034 //----------------------------------------------------------------------
430035 // The structured type 'inode_t' must be pre-declared here, because
430036 // the type sb_t, described before the inode structure, has a member
430037 // pointing to a type 'inode_t'. So, must be declared previously
430038 // the type 'inode_t' as made of a type 'struct inode', then the
430039 // structure 'inode' can be described. But for a matter of coherence,
430040 // all other structured data declared inside this file follow the
430041 // same procedure.
430042 //
430043 typedef struct sb        sb_t;
430044 typedef struct inode     inode_t;
430045 typedef struct file      file_t;
430046 typedef struct fd        fd_t;
430047 typedef struct directory directory_t;
430048 //----------------------------------------------------------------------
430049 #define SB_MAX_SLOTS  2                 // Handle max 2 file systems.
430050
430051 struct sb {                             // File system super block:
430052     uint16_t      inodes;               // inodes available;
430053     uint16_t      zones;                // zones available (disk size);
430054     uint16_t      map_inode_blocks;     // inode bit map blocks;
430055     uint16_t      map_zone_blocks;      // data-zone bit map blocks;
430056     uint16_t      first_data_zone;      // first data-zone;
430057     uint16_t      log2_size_zone;       // log_2 (size_zone/block_size);
430058     uint32_t      max_file_size;        // max file size in bytes;
430059     uint16_t      magic_number;         // file system magic number.
430060     //------------------------------------------------------------------
430061     // Extra management data, not saved inside the file system
430062     // super block.
430063     //------------------------------------------------------------------
430064     dev_t         device;                       // FS device [3]
430065     inode_t      *inode_mounted_on;             // [4]
430066     blksize_t     blksize;                      // Calculated zone size.
430067     int           options;                      // [5]
430068     uint16_t      map_inode[SB_MAP_INODE_SIZE];
430069     uint16_t      map_zone[SB_MAP_ZONE_SIZE];
430070     char          changed;
430071 };
```

```
430072
430073 extern sb_t    sb_table[SB_MAX_SLOTS];
430074 //
430075 // [3] the member 'device' must be kept at the same position, because
430076 //     it is used to calculate the super block header size, saved on
430077 //     disk.
430078 //
430079 // [4] If this pointer is not NULL, the super block is related to a
430080 //     device mounted on a directory. The inode of such directory is
430081 //     recorded here. Please note that it is type 'void *', instead of
430082 //     type 'inode_t', because type 'inode_t' is declared after type
430083 //     'sb_t'.
430084 //     Please note that the type 'sb_t' is declared before the
430085 //     type 'inode_t', but this member points to a type 'inode_t'.
430086 //     This is the reason because it was necessary to declare first
430087 //     the type 'inode_t' as made of 'struct inode', to be described
430088 //     later. For coherence, all derived type made of structured data,
430089 //     are first declared as structure, and then, later, described.
430090 //
430091 // [5] Mount options can be only 'MOUNT_DEFAULT' or 'MOUNT_RO',
430092 //     as defined inside file 'lib/sys/os16.h'.
430093 //
430094 //----------------------------------------------------------------------
430095 #define INODE_MAX_SLOTS           (8 * OPEN_MAX)
430096
430097 struct inode {                              // Inode (32 byte total):
430098     mode_t        mode;                     // file type and permissions;
430099     uid_t         uid;                      // user ID (16 bit);
430100     ssize_t       size;                     // file size in bytes;
430101     time_t        time;                     // file data modification time;
430102     uint8_t       gid;                      // group ID (8 bit);
430103     uint8_t       links;                    // links to the inode;
430104     zno_t         direct[7];                // direct zones;
430105     zno_t         indirect1;                // indirect zones;
430106     zno_t         indirect2;                // double indirect zones.
430107     //------------------------------------------------------------------
430108     // Extra management data, not saved inside the disk file system.
430109     //------------------------------------------------------------------
430110     sb_t         *sb;                       // Inode's super block. [7]
430111     ino_t         ino;                      // Inode number.
430112     sb_t         *sb_attached;              // [8]
430113     blkcnt_t      blkcnt;                   // Rounded size/blksize.
430114     unsigned char references;               // Run time active references.
430115     char          changed;                  // 1 == to be saved.
430116 };
430117
430118 extern inode_t inode_table[INODE_MAX_SLOTS];
430119 //
```

```
430120 // [7] the member 'sb' must be kept at the same position, because
430121 //     it is used to calculate the inode header size, saved on disk.
430122 //
430123 // [8] If the inode is a mount point for another device, the other
430124 //     super block pointer is saved inside 'sb_attached'.
430125 //
430126 //----------------------------------------------------------------------
430127 #define FILE_MAX_SLOTS            (16 * OPEN_MAX)
430128 
430129 struct file {
430130     int           references;
430131     off_t         offset;              // File position.
430132     int           oflags;              // Open mode: r/w/r+w [9]
430133     inode_t      *inode;
430134 };
430135 
430136 extern file_t file_table[FILE_MAX_SLOTS];
430137 //
430138 // [9] the member 'oflags' can get only O_RDONLY, O_WRONLY, O_RDWR,
430139 //     (from header 'fcntl.h') combined with OR binary operator.
430140 //
430141 //----------------------------------------------------------------------
430142 struct fd {
430143     int           fl_flags;            // File status flags and file
430144                                        // access modes. [10]
430145     int           fd_flags;            // File descriptor flags:
430146                                        // currently only FD_CLOEXEC.
430147     file_t       *file;                // Pointer to the file table.
430148 };
430149 //
430150 // [10] the member 'fl_flags' can get only O_RDONLY, O_WRONLY, O_RDWR,
430151 //      O_CREAT, O_EXCL, O_NOCTTY, O_TRUNC and O_APPEND
430152 //      (from header 'fcntl.h') combined with OR binary
430153 //      operator. Options like O_DSYNC, O_NONBLOCK, O_RSYNC and O_SYNC
430154 //      are not taken into consideration by os16.
430155 //
430156 //----------------------------------------------------------------------
430157 struct directory {                     // Directory entry:
430158     ino_t         ino;                 // inode number;
430159     char          name[NAME_MAX];      // file name.
430160 };
430161 //----------------------------------------------------------------------
430162 sb_t    *sb_reference    (dev_t device);
430163 sb_t    *sb_mount        (dev_t device, inode_t **inode_mnt,
430164                           int options);
430165 sb_t    *sb_get          (dev_t device, sb_t *sb);
430166 int      sb_save         (sb_t *sb);
430167 int      sb_zone_status  (sb_t *sb, zno_t zone);
```

```
430168 int      sb_inode_status (sb_t *sb, ino_t ino);
430169 //----------------------------------------------------------------------
430170 zno_t    zone_alloc           (sb_t *sb);
430171 int      zone_free            (sb_t *sb, zno_t zone);
430172 int      zone_read            (sb_t *sb, zno_t zone, void *buffer);
430173 int      zone_write           (sb_t *sb, zno_t zone, void *buffer);
430174 //----------------------------------------------------------------------
430175 inode_t *inode_reference      (dev_t device, ino_t ino);
430176 inode_t *inode_alloc          (dev_t device, mode_t mode, uid_t uid);
430177 int      inode_free           (inode_t *inode);
430178 inode_t *inode_get            (dev_t device, ino_t ino);
430179 int      inode_save           (inode_t *inode);
430180 int      inode_put            (inode_t *inode);
430181 int      inode_truncate       (inode_t *inode);
430182 zno_t    inode_zone           (inode_t *inode, zno_t fzone, int write);
430183 inode_t *inode_stdio_dev_make (dev_t device, mode_t mode);
430184 blkcnt_t inode_fzones_read    (inode_t *inode, zno_t zone_start,
430185                                void *buffer, blkcnt_t blkcnt);
430186 blkcnt_t inode_fzones_write   (inode_t *inode, zno_t zone_start,
430187                                void *buffer, blkcnt_t blkcnt);
430188 ssize_t  inode_file_read      (inode_t *inode, off_t offset,
430189                                void *buffer, size_t count, int *eof);
430190 ssize_t  inode_file_write     (inode_t *inode, off_t offset,
430191                                void *buffer, size_t count);
430192 int      inode_check          (inode_t *inode, mode_t type,
430193                                int perm, uid_t uid);
430194 int      inode_dir_empty      (inode_t *inode);
430195 //----------------------------------------------------------------------
430196 file_t  *file_reference       (int fno);
430197 file_t  *file_stdio_dev_make  (dev_t device, mode_t mode, int oflags);
430198 //----------------------------------------------------------------------
430199 inode_t *path_inode      (pid_t pid, const char *path);
430200 int      path_chdir      (pid_t pid, const char *path);
430201 dev_t    path_device     (pid_t pid, const char *path);
430202 int      path_full       (const char *path,
430203                           const char *path_cwd,
430204                           char *full_path);
430205 int      path_fix        (char *path);
430206 inode_t *path_inode_link (pid_t pid, const char *path, inode_t *inode,
430207                           mode_t mode);
430208 int      path_link       (pid_t pid, const char *path_old,
430209                           const char *path_new);
430210 int      path_mkdir      (pid_t pid, const char *path, mode_t mode);
430211 int      path_mknod      (pid_t pid, const char *path, mode_t mode,
430212                           dev_t device);
430213 int      path_mount      (pid_t pid, const char *path_dev,
430214                           const char *path_mnt,
430215                           int options);
```

```
430216 int      path_umount     (pid_t pid, const char *path_mnt);
430217 int      path_stat       (pid_t pid, const char *path,
430218                           struct stat *buffer);
430219 int      path_chmod      (pid_t pid, const char *path, mode_t mode);
430220 int      path_chown      (pid_t pid, const char *path, uid_t uid,
430221                           gid_t gid);
430222 int      path_unlink     (pid_t pid, const char *path);
430223 //----------------------------------------------------------------------
430224 fd_t    *fd_reference (pid_t pid, int *fdn);
430225 int      fd_chmod     (pid_t pid, int fdn, mode_t mode);
430226 int      fd_chown     (pid_t pid, int fdn, uid_t uid, gid_t gid);
430227 int      fd_close     (pid_t pid, int fdn);
430228 int      fd_fcntl     (pid_t pid, int fdn, int cmd, int arg);
430229 int      fd_dup       (pid_t pid, int fdn_old, int fdn_min);
430230 int      fd_dup2      (pid_t pid, int fdn_old, int fdn_new);
430231 off_t    fd_lseek     (pid_t pid, int fdn, off_t offset, int whence);
430232 int      fd_open      (pid_t pid, const char *path, int oflags,
430233                        mode_t mode);
430234 ssize_t  fd_read      (pid_t pid, int fdn, void *buffer, size_t count,
430235                        int *eof);
430236 int      fd_stat      (pid_t pid, int fdn, struct stat *buffer);
430237 ssize_t  fd_write     (pid_t pid, int fdn, const void *buffer,
430238                        size_t count);
430239 //----------------------------------------------------------------------
430240
430241 #endif
```

100.4.1 kernel/fs/fd_chmod.c

Si veda la sezione 99.3.1.

```
440001 #include <kernel/proc.h>
440002 #include <kernel/k_libc.h>
440003 #include <sys/stat.h>
440004 #include <errno.h>
440005 //----------------------------------------------------------------------
440006 int
440007 fd_chmod (pid_t pid, int fdn, mode_t mode)
440008 {
440009     proc_t   *ps;
440010     inode_t  *inode;
440011     //
440012     // Get process.
440013     //
440014     ps = proc_reference (pid);
440015     //
440016     // Verify if the file descriptor is valid.
440017     //
```

```
440018     if (ps->fd[fdn].file == NULL)
440019       {
440020         errset (EBADF);          // Bad file descriptor.
440021         return (-1);
440022       }
440023     //
440024     // Reach the inode.
440025     //
440026     inode = ps->fd[fdn].file->inode;
440027     //
440028     // Verify to be the owner, or at least to be UID == 0.
440029     //
440030     if (ps->euid != inode->uid && ps->euid != 0)
440031       {
440032         errset (EACCES);         // Permission denied.
440033         return (-1);
440034       }
440035     //
440036     // Update the mode: the file type is kept and the
440037     // rest is taken form the parameter 'mode'.
440038     //
440039     inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
440040     //
440041     // Save the inode.
440042     //
440043     inode->changed = 1;
440044     inode_save (inode);
440045     //
440046     // Return.
440047     //
440048     return (0);
440049 }
```

100.4.2 kernel/fs/fd_chown.c

«

Si veda la sezione 99.3.2.

```
450001 #include <kernel/proc.h>
450002 #include <kernel/k_libc.h>
450003 #include <errno.h>
450004 //----------------------------------------------------------------------
450005 int
450006 fd_chown (pid_t pid, int fdn, uid_t uid, gid_t gid)
450007 {
450008     proc_t   *ps;
450009     inode_t  *inode;
450010     //
450011     // Get process.
```

```
450012      //
450013      ps = proc_reference (pid);
450014      //
450015      // Verify if the file descriptor is valid.
450016      //
450017      if (ps->fd[fdn].file == NULL)
450018        {
450019          errset (EBADF);          // Bad file descriptor.
450020          return (-1);
450021        }
450022      //
450023      // Reach the inode.
450024      //
450025      inode = ps->fd[fdn].file->inode;
450026      //
450027      // Verify to be root, as the ability to change group
450028      // is not taken into consideration.
450029      //
450030      if (ps->euid != 0)
450031        {
450032          errset (EACCES);         // Permission denied.
450033          return (-1);
450034        }
450035      //
450036      // Update the ownership.
450037      //
450038      if (uid != -1)
450039        {
450040          inode->uid     = uid;
450041          inode->changed = 1;
450042        }
450043      if (gid != -1)
450044        {
450045          inode->gid     = gid;
450046          inode->changed = 1;
450047        }
450048      //
450049      // Save the inode.
450050      //
450051      inode->changed = 1;
450052      inode_save (inode);
450053      //
450054      // Return.
450055      //
450056      return (0);
450057  }
```

100.4.3 kernel/fs/fd_close.c

«

Si veda la sezione 99.3.3.

```
460001 #include <kernel/proc.h>
460002 #include <kernel/k_libc.h>
460003 #include <errno.h>
460004 //----------------------------------------------------------------------
460005 int
460006 fd_close (pid_t pid, int fdn)
460007 {
460008     inode_t            *inode;
460009     file_t             *file;
460010     fd_t *fd;
460011     //
460012     // Get file descriptor.
460013     //
460014     fd = fd_reference (pid, &fdn);
460015     if (fd == NULL                ||
460016         fd->file == NULL          ||
460017         fd->file->inode == NULL )
460018       {
460019         errset (EBADF);           // Bad file descriptor.
460020         return (-1);
460021       }
460022     //
460023     // Get file.
460024     //
460025     file = fd->file;
460026     //
460027     // Get inode.
460028     //
460029     inode = file->inode;
460030     //
460031     // Reduce references inside the file table item
460032     // and remove item if it reaches zero.
460033     //
460034     file->references--;
460035     if (file->references == 0)
460036       {
460037         file->oflags = 0;
460038         file->inode  = NULL;
460039         //
460040         // Put inode, because there are no more file references.
460041         //
460042         inode_put (inode);
460043       }
460044     //
460045     // Remove file descriptor.
```

```
460046      //
460047      fd->fl_flags = 0;
460048      fd->fd_flags = 0;
460049      fd->file   = NULL;
460050      //
460051      //
460052      //
460053      return (0);
460054  }
```

100.4.4 kernel/fs/fd_dup.c

«

Si veda la sezione 99.3.4.

```
470001  #include <kernel/proc.h>
470002  #include <kernel/k_libc.h>
470003  #include <errno.h>
470004  #include <fcntl.h>
470005  //----------------------------------------------------------------------
470006  int
470007  fd_dup (pid_t pid, int fdn_old, int fdn_min)
470008  {
470009      proc_t *ps;
470010      int     fdn_new;
470011      //
470012      // Verify argument.
470013      //
470014      if (fdn_min < 0 || fdn_min >= OPEN_MAX)
470015        {
470016          errset (EINVAL);                  // Invalid argument.
470017          return (-1);
470018        }
470019      //
470020      // Get process.
470021      //
470022      ps = proc_reference (pid);
470023      //
470024      // Verify if 'fdn_old' is a valid value.
470025      //
470026      if (fdn_old < 0                        ||
470027          fdn_old >= OPEN_MAX                ||
470028          ps->fd[fdn_old].file == NULL)
470029        {
470030          errset (EBADF);                   // Bad file descriptor.
470031          return (-1);
470032        }
470033      //
470034      // Find the first free slot and duplicate the file descriptor.
```

```
470035     //
470036     for (fdn_new = fdn_min; fdn_new < OPEN_MAX; fdn_new++)
470037       {
470038         if (ps->fd[fdn_new].file == NULL)
470039           {
470040             ps->fd[fdn_new].fl_flags = ps->fd[fdn_old].fl_flags;
470041             ps->fd[fdn_new].fd_flags
470042                 = ps->fd[fdn_old].fd_flags & ~FD_CLOEXEC;
470043             ps->fd[fdn_new].file = ps->fd[fdn_old].file;
470044             ps->fd[fdn_new].file->references++;
470045             return (fdn_new);
470046           }
470047       }
470048     //
470049     // No fd slot available.
470050     //
470051     errset (EMFILE);            // Too many open files.
470052     return (-1);
470053 }
```

100.4.5 kernel/fs/fd_dup2.c

Si veda la sezione 99.3.4.

```
480001 #include <kernel/proc.h>
480002 #include <kernel/k_libc.h>
480003 #include <errno.h>
480004 #include <fcntl.h>
480005 //----------------------------------------------------------------------
480006 int
480007 fd_dup2 (pid_t pid, int fdn_old, int fdn_new)
480008 {
480009     proc_t *ps;
480010     int     status;
480011     //
480012     // Get process.
480013     //
480014     ps = proc_reference (pid);
480015     //
480016     // Verify if 'fdn_old' is a valid value.
480017     //
480018     if (fdn_old < 0                     ||
480019         fdn_old >= OPEN_MAX             ||
480020         ps->fd[fdn_old].file == NULL)
480021       {
480022         errset (EBADF);                 // Bad file descriptor.
480023         return (-1);
480024       }
```

```
480025      //
480026      // Check if 'fd_old' and 'fd_new' are the same.
480027      //
480028      if (fdn_old == fdn_new)
480029        {
480030          return (fdn_new);
480031        }
480032      //
480033      // Close 'fdn_new' if it is open and copy 'fdn_old' into it.
480034      //
480035      if (ps->fd[fdn_new].file != NULL)
480036        {
480037          status = fd_close (pid, fdn_new);
480038          if (status != 0)
480039            {
480040              return (-1);
480041            }
480042        }
480043      ps->fd[fdn_new].fl_flags = ps->fd[fdn_old].fl_flags;
480044      ps->fd[fdn_new].fd_flags = ps->fd[fdn_old].fd_flags & ~FD_CLOEXEC;
480045      ps->fd[fdn_new].file = ps->fd[fdn_old].file;
480046      ps->fd[fdn_new].file->references++;
480047      return (fdn_new);
480048  }
```

100.4.6 kernel/fs/fd_fcntl.c

Si veda la sezione 99.3.6.

```
490001  #include <kernel/proc.h>
490002  #include <kernel/k_libc.h>
490003  #include <errno.h>
490004  #include <fcntl.h>
490005  //----------------------------------------------------------------------
490006  int
490007  fd_fcntl (pid_t pid, int fdn, int cmd, int arg)
490008  {
490009      proc_t   *ps;
490010      inode_t  *inode;
490011      int       mask;
490012      //
490013      // Get process.
490014      //
490015      ps = proc_reference (pid);
490016      //
490017      // Verify if the file descriptor is valid.
490018      //
490019      if (ps->fd[fdn].file == NULL)
```

```
490020          {
490021            errset (EBADF);                  // Bad file descriptor.
490022            return (-1);
490023          }
490024        //
490025        // Reach the inode.
490026        //
490027        inode = ps->fd[fdn].file->inode;
490028        //
490029        //
490030        //
490031        switch (cmd)
490032          {
490033            case F_DUPFD:
490034                return (fd_dup (pid, fdn, arg));
490035                break;
490036            case F_GETFD:
490037                return (ps->fd[fdn].fd_flags);
490038                break;
490039            case F_SETFD:
490040                ps->fd[fdn].fd_flags = arg;
490041                return (0);
490042            case F_GETFL:
490043                return (ps->fd[fdn].fl_flags);
490044            case F_SETFL:
490045                //
490046                // Calculate a mask with bits that are not to be set.
490047                //
490048                mask = (O_ACCMODE
490049                       | O_CREAT
490050                       | O_EXCL
490051                       | O_NOCTTY
490052                       | O_TRUNC);
490053                //
490054                // Set to zero the bits that are not to be set from
490055                // the argument.
490056                //
490057                arg = (arg & ~mask);
490058                //
490059                // Set to zero the bit that *are* to be set.
490060                //
490061                ps->fd[fdn].fl_flags &= mask;
490062                //
490063                // Set the bits, already filtered inside the argument.
490064                //
490065                ps->fd[fdn].fl_flags |= arg;
490066                //
490067                return (0);
```

```
490068         default:
490069             errset (EINVAL);                    // Not implemented.
490070             return (-1);
490071       }
490072 }
```

100.4.7 kernel/fs/fd_lseek.c

Si veda la sezione 99.3.7.

```
500001 #include <kernel/proc.h>
500002 #include <kernel/k_libc.h>
500003 #include <errno.h>
500004 //----------------------------------------------------------------------
500005 off_t
500006 fd_lseek (pid_t pid, int fdn, off_t offset, int whence)
500007 {
500008     inode_t           *inode;
500009     file_t            *file;
500010     fd_t *fd;
500011     off_t              test_offset;
500012     //
500013     // Get file descriptor.
500014     //
500015     fd = fd_reference (pid, &fdn);
500016     if (fd == NULL                ||
500017         fd->file == NULL          ||
500018         fd->file->inode == NULL )
500019       {
500020         errset (EBADF);                 // Bad file descriptor.
500021         return (-1);
500022       }
500023     //
500024     // Get file table item.
500025     //
500026     file = fd->file;
500027     //
500028     // Get inode.
500029     //
500030     inode = file->inode;
500031     //
500032     // Change position depending on the 'whence' parameter.
500033     //
500034     if (whence == SEEK_SET)
500035       {
500036         if (offset < 0)
500037           {
500038             errset (EINVAL);                    // Invalid argument.
```

```
500039                return ((off_t) -1);
500040              }
500041            else
500042              {
500043                fd->file->offset = offset;
500044              }
500045          }
500046        else if (whence == SEEK_CUR)
500047          {
500048            test_offset  = fd->file->offset;
500049            test_offset += offset;
500050            if (test_offset < 0)
500051              {
500052                errset (EINVAL);                       // Invalid argument.
500053                return ((off_t) -1);
500054              }
500055            else
500056              {
500057                fd->file->offset = test_offset;
500058              }
500059          }
500060        else if (whence == SEEK_END)
500061          {
500062            test_offset  = inode->size;
500063            test_offset += offset;
500064            if (test_offset < 0)
500065              {
500066                errset (EINVAL);                       // Invalid argument.
500067                return ((off_t) -1);
500068              }
500069            else
500070              {
500071                fd->file->offset = test_offset;
500072              }
500073          }
500074        else
500075          {
500076            errset (EINVAL);                           // Invalid argument.
500077            return ((off_t) -1);
500078          }
500079        //
500080        // Return the new file position.
500081        //
500082        return (fd->file->offset);
500083  }
```

100.4.8 kernel/fs/fd_open.c

Si veda la sezione 99.3.8.

```
510001 #include <kernel/proc.h>
510002 #include <kernel/k_libc.h>
510003 #include <errno.h>
510004 #include <fcntl.h>
510005 //----------------------------------------------------------------------
510006 int
510007 fd_open (pid_t pid, const char *path, int oflags, mode_t mode)
510008 {
510009     proc_t   *ps;
510010     inode_t  *inode;
510011     int       status;
510012     file_t   *file;
510013     fd_t     *fd;
510014     int       fdn;
510015     char      full_path[PATH_MAX];
510016     int       perm;
510017     tty_t    *tty;
510018     mode_t    umask;
510019     int       errno_save;
510020     //
510021     // Get process.
510022     //
510023     ps = proc_reference (pid);
510024     //
510025     // Correct the mode with the umask. As it is not a directory, to the
510026     // mode are removed execution and sticky permissions.
510027     //
510028     umask = ps->umask | 01111;
510029     mode &= ~umask;
510030     //
510031     // Check open options.
510032     //
510033     if (oflags & O_WRONLY)
510034       {
510035         //
510036         // The file is to be opened for write, or for read/write.
510037         // Try to get inode.
510038         //
510039         inode = path_inode (pid, path);
510040         if (inode == NULL)
510041           {
510042             //
510043             // Cannot get the inode. See if there is the creation
510044             // option.
510045             //
```

```
510046                if (oflags & O_CREAT)
510047                  {
510048                    //
510049                    // Try to create the missing inode: the file must be a
510050                    // regular one, so add the mode.
510051                    //
510052                    path_full (path, ps->path_cwd, full_path);
510053                    inode = path_inode_link (pid, full_path, NULL,
510054                                             (mode | S_IFREG));
510055                    if (inode == NULL)
510056                      {
510057                        //
510058                        // Sorry: cannot create the inode! Variable 'errno'
510059                        // is already set by 'path_inode_link()'.
510060                        //
510061                        errset (errno);
510062                        return (-1);
510063                      }
510064                  }
510065                else
510066                  {
510067                    //
510068                    // Cannot open the inode. Variable 'errno'
510069                    // should be already set by 'path_inode()'.
510070                    //
510071                    errset (errno);
510072                    return (-1);
510073                  }
510074              }
510075            //
510076            // The inode was read or created: check if it must be
510077            // truncated. It can be truncated only if it is a regular
510078            // file.
510079            //
510080            if (oflags & O_TRUNC && inode->mode & S_IFREG)
510081              {
510082                //
510083                // Truncate inode.
510084                //
510085                status = inode_truncate (inode);
510086                if (status != 0)
510087                  {
510088                    //
510089                    // Cannot truncate the inode: release it and return.
510090                    // But this error should never happen, because the
510091                    // function 'inode_truncate()' will not return any
510092                    // other value than zero.
510093                    //
```

```
510094                  errno_save = errno;
510095                  inode_put (inode);
510096                  errset (errno_save);
510097                  return (-1);
510098                }
510099            }
510100        }
510101      else
510102        {
510103          //
510104          // The file is to be opened for read, but not for write.
510105          // Try to get inode.
510106          //
510107          inode = path_inode (pid, path);
510108          if (inode == NULL)
510109            {
510110              //
510111              // Cannot open the file.
510112              //
510113              errset (errno);
510114              return (-1);
510115            }
510116        }
510117      //
510118      // An inode was opened: check type and access permissions.
510119      // All file types are good, even directories, as the type
510120      // DIR is implemented through file descriptors.
510121      //
510122      perm = 0;
510123      if (oflags & O_RDONLY) perm |= 4;
510124      if (oflags & O_WRONLY) perm |= 2;
510125      status = inode_check (inode, S_IFMT, perm, ps->uid);
510126      if (status != 0)
510127        {
510128          //
510129          // The file type is not correct or the user does not have
510130          // permissions.
510131          //
510132          return (-1);
510133        }
510134      //
510135      // Allocate the file, inside the file table.
510136      //
510137      file = file_reference (-1);
510138      if (file == NULL)
510139        {
510140          //
510141          // Cannot allocate the file inside the file table: release the
```

```
510142            // inode, update 'errno' and return.
510143            //
510144            inode_put (inode);
510145            errset (ENFILE);        // Too many files open in system.
510146            return (-1);
510147          }
510148        //
510149        // Put some data inside the file item. Only options
510150        // O_RDONLY and O_WRONLY are kept here, because the O_APPEND
510151        // is saved inside the file descriptor table.
510152        //
510153        file->references = 1;
510154        file->oflags     = (oflags & (O_RDONLY | O_WRONLY));
510155        file->inode      = inode;
510156        //
510157        // Allocate the file descriptor: variable 'fdn' will be modified
510158        // by the call to 'fd_reference()'.
510159        //
510160        fdn = -1;
510161        fd = fd_reference (pid, &fdn);
510162        if (fd == NULL)
510163          {
510164            //
510165            // Cannot allocate the file descriptor: remove the item from
510166            // file table.
510167            //
510168            file->references = 0;
510169            file->oflags     = 0;
510170            file->inode      = NULL;
510171            //
510172            // Release the inode.
510173            //
510174            inode_put (inode);
510175            //
510176            // Return an error.
510177            //
510178            errset (EMFILE);        // Too many open files.
510179            return (-1);
510180          }
510181        //
510182        // File descriptor allocated: put some data inside the
510183        // file descriptor item.
510184        //
510185        fd->fl_flags = (oflags & (O_RDONLY | O_WRONLY | O_APPEND));
510186        fd->fd_flags = 0;
510187        fd->file = file;
510188        fd->file->offset = 0;
510189        //
```

```
510190      // Check if it is a terminal (currently only consoles), if it is
510191      // opened for read and write, and if it have to be set as the
510192      // controlling terminal. This thing is done here because there is
510193      // not a real device driver.
510194      //
510195      if ((S_ISCHR (inode->mode)) &&
510196          (oflags & O_RDONLY)     &&
510197          (oflags & O_WRONLY))
510198        {
510199          //
510200          // The inode is a character special file (related to a character
510201          // device), opened for read and write!
510202          //
510203          if ((inode->direct[0] & 0xFF00) == (DEV_CONSOLE_MAJOR << 8))
510204            {
510205              //
510206              // It is a terminal (currently only consoles are possible).
510207              // Get the tty reference.
510208              //
510209              tty = tty_reference ((dev_t) inode->direct[0]);
510210              //
510211              // Verify that the terminal is not already the controlling
510212              // terminal of some process group.
510213              //
510214              if (tty->pgrp == 0)
510215                {
510216                  //
510217                  // The terminal is free: verify if the current process
510218                  // needs a controlling terminal.
510219                  //
510220                  if (ps->device_tty == 0 && ps->pgrp == pid)
510221                    {
510222                      //
510223                      // It is a group leader with no controlling
510224                      // terminal: set the controlling terminal.
510225                      //
510226                      ps->device_tty = inode->direct[0];
510227                      tty->pgrp      = ps->pgrp;
510228                    }
510229                }
510230            }
510231        }
510232      //
510233      // Return the file descriptor.
510234      //
510235      return (fdn);
510236  }
```

100.4.9 kernel/fs/fd_read.c

«

Si veda la sezione 99.3.9.

```
520001 #include <kernel/proc.h>
520002 #include <kernel/k_libc.h>
520003 #include <errno.h>
520004 #include <fcntl.h>
520005 //----------------------------------------------------------------------
520006 ssize_t
520007 fd_read (pid_t pid, int fdn, void *buffer, size_t count, int *eof)
520008 {
520009     fd_t  *fd;
520010     ssize_t  size_read;
520011     //
520012     // Get file descriptor.
520013     //
520014     fd = fd_reference (pid, &fdn);
520015     if (fd                == NULL   ||
520016         fd->file          == NULL   ||
520017         fd->file->inode == NULL )
520018       {
520019         errset (EBADF);                    // Bad file descriptor.
520020         return ((ssize_t) -1);
520021       }
520022     //
520023     // Check if it is opened for read.
520024     //
520025     if (!(fd->file->oflags & O_RDONLY))
520026       {
520027         //
520028         // The file is not opened for read.
520029         //
520030         errset (EINVAL);                   // Invalid argument.
520031         return ((ssize_t) -1);
520032       }
520033     //
520034     // It is not a mistake to read a directory, as 'dirent.h' is
520035     // implemented through file descriptors.
520036     //
520037     //  //
520038     //  // Check if it is a directory.
520039     //  //
520040     //  if (fd->file->inode->mode & S_IFDIR)
520041     //    {
520042     //      errset (EISDIR);                   // Is a directory.
520043     //      return ((ssize_t) -1);
520044     //    }
520045     //
```

```
520046      // Check the kind of file to be read and read it.
520047      //
520048      if   (S_ISBLK (fd->file->inode->mode)
520049        || S_ISCHR (fd->file->inode->mode))
520050        {
520051          //
520052          // A device is to be read.
520053          //
520054          size_read = dev_io (pid, (dev_t) fd->file->inode->direct[0],
520055                              DEV_READ, fd->file->offset, buffer, count,
520056                              eof);
520057        }
520058      else if (S_ISREG (fd->file->inode->mode))
520059        {
520060          //
520061          // A regular file is to be read.
520062          //
520063          size_read = inode_file_read (fd->file->inode, fd->file->offset,
520064                                       buffer, count, eof);
520065        }
520066      else if (S_ISDIR (fd->file->inode->mode))
520067        {
520068          //
520069          // A directory, is to be read.
520070          //
520071          size_read = inode_file_read (fd->file->inode, fd->file->offset,
520072                                       buffer, count, eof);
520073        }
520074      else
520075        {
520076          //
520077          // Unsupported file type.
520078          //
520079          errset (E_FILE_TYPE_UNSUPPORTED);       //File type unsupported.
520080          return ((ssize_t) -1);
520081        }
520082      //
520083      // Update the file descriptor internal offset.
520084      //
520085      if (size_read > 0)
520086        {
520087          fd->file->offset += size_read;
520088        }
520089      //
520090      // Just return the size read, even if it is an error. Please note
520091      // that a size of zero might tell that it is the end of file, or
520092      // just that the read should be retried.
520093      //
```

```
520094     return (size_read);
520095 }
```

100.4.10 kernel/fs/fd_reference.c

«

Si veda la sezione 99.3.10.

```
530001 #include <kernel/proc.h>
530002 #include <kernel/k_libc.h>
530003 #include <errno.h>
530004 //----------------------------------------------------------------------
530005 fd_t *
530006 fd_reference (pid_t pid, int *fdn)
530007 {
530008     proc_t *ps;
530009     //
530010     // Get process.
530011     //
530012     ps = proc_reference (pid);
530013     //
530014     // See what to do.
530015     //
530016     if (*fdn < 0)
530017       {
530018         //
530019         // Find the first free slot.
530020         //
530021         for (*fdn = 0; *fdn < OPEN_MAX; (*fdn)++)
530022           {
530023             if (ps->fd[*fdn].file == NULL)
530024               {
530025                 return (&(ps->fd[*fdn]));
530026               }
530027           }
530028         *fdn = -1;
530029         return (NULL);
530030       }
530031     else
530032       {
530033         if (*fdn < OPEN_MAX)
530034           {
530035             //
530036             // Might return even a free file descriptor.
530037             //
530038             return (&(ps->fd[*fdn]));
530039           }
530040         else
530041           {
```

```
530042                 return (NULL);
530043             }
530044         }
530045 }
```

100.4.11 kernel/fs/fd_stat.c

Si veda la sezione 99.3.50.

```
540001 #include <kernel/proc.h>
540002 #include <kernel/k_libc.h>
540003 #include <errno.h>
540004 #include <fcntl.h>
540005 //----------------------------------------------------------------------
540006 int
540007 fd_stat (pid_t pid, int fdn, struct stat *buffer)
540008 {
540009     proc_t   *ps;
540010     inode_t  *inode;
540011     //
540012     // Get process.
540013     //
540014     ps = proc_reference (pid);
540015     //
540016     // Verify if the file descriptor is valid.
540017     //
540018     if (ps->fd[fdn].file == NULL)
540019       {
540020         errset (EBADF);                 // Bad file descriptor.
540021         return (-1);
540022       }
540023     //
540024     // Reach the inode.
540025     //
540026     inode = ps->fd[fdn].file->inode;
540027     //
540028     // Inode loaded: update the buffer.
540029     //
540030     buffer->st_dev      = inode->sb->device;
540031     buffer->st_ino      = inode->ino;
540032     buffer->st_mode     = inode->mode;
540033     buffer->st_nlink    = inode->links;
540034     buffer->st_uid      = inode->uid;
540035     buffer->st_gid      = inode->gid;
540036     if (S_ISBLK (buffer->st_mode) || S_ISCHR (buffer->st_mode))
540037       {
540038         buffer->st_rdev = inode->direct[0];
540039       }
```

```
540040      else
540041        {
540042          buffer->st_rdev = 0;
540043        }
540044      buffer->st_size      = inode->size;
540045      buffer->st_atime     = inode->time;  // All times are the same for
540046      buffer->st_mtime     = inode->time;  // Minix 1 file system.
540047      buffer->st_ctime     = inode->time;  //
540048      buffer->st_blksize   = inode->sb->blksize;
540049      buffer->st_blocks    = inode->blkcnt;
540050      //
540051      // If the inode is a device special file, the 'st_rdev' value is
540052      // taken from the first direct zone (as of Minix 1 organization).
540053      //
540054      if (S_ISBLK(inode->mode) || S_ISCHR(inode->mode))
540055        {
540056          buffer->st_rdev = inode->direct[0];
540057        }
540058      else
540059        {
540060          buffer->st_rdev = 0;
540061        }
540062      //
540063      // Return.
540064      //
540065      return (0);
540066  }
```

100.4.12 kernel/fs/fd_write.c

Si veda la sezione 99.3.12.

```
550001  #include <kernel/proc.h>
550002  #include <kernel/k_libc.h>
550003  #include <errno.h>
550004  #include <fcntl.h>
550005  //----------------------------------------------------------------------
550006  ssize_t
550007  fd_write (pid_t pid, int fdn, const void *buffer, size_t count)
550008  {
550009      proc_t   *ps;
550010      fd_t     *fd;
550011      ssize_t  size_written;
550012      //
550013      // Get process.
550014      //
550015      ps = proc_reference (pid);
550016      //
```

```
550017      // Get file descriptor.
550018      //
550019      fd = fd_reference (pid, &fdn);
550020      if (fd              == NULL   ||
550021          fd->file        == NULL   ||
550022          fd->file->inode == NULL )
550023        {
550024          //
550025          // The file descriptor pointer is not valid.
550026          //
550027          errset (EBADF);                  // Bad file descriptor.
550028          return ((ssize_t) -1);
550029        }
550030      //
550031      // Check if it is opened for write.
550032      //
550033      if (!(fd->file->oflags & O_WRONLY))
550034        {
550035          //
550036          // The file is not opened for write.
550037          //
550038          errset (EINVAL);                 // Invalid argument.
550039          return ((ssize_t) -1);
550040        }
550041      //
550042      // Check if it is a directory.
550043      //
550044      if (fd->file->inode->mode & S_IFDIR)
550045        {
550046          errset (EISDIR);                 // Is a directory.
550047          return ((ssize_t) -1);
550048        }
550049      //
550050      // It should be a valid type of file to be written. Check if it is
550051      // opened in append mode: if so, must move the write offset to the
550052      // end.
550053      //
550054      if (fd->fl_flags & O_APPEND)
550055        {
550056          fd->file->offset = fd->file->inode->size;
550057        }
550058      //
550059      // Check the kind of file to be written and write it.
550060      //
550061      if (fd->file->inode->mode & S_IFBLK ||
550062          fd->file->inode->mode & S_IFCHR)
550063        {
550064          //
```

```
550065          // A device is to be written.
550066          //
550067          size_written = dev_io (pid, (dev_t) fd->file->inode->direct[0],
550068                                 DEV_WRITE, fd->file->offset, buffer,
550069                                 count, NULL);
550070        }
550071      else if (fd->file->inode->mode & S_IFREG)
550072        {
550073          //
550074          // A regular file is to be written.
550075          //
550076          size_written = inode_file_write (fd->file->inode,
550077                                           fd->file->offset,
550078                                           buffer, count);
550079        }
550080      else
550081        {
550082          //
550083          // Unsupported file type.
550084          //
550085          errset (E_FILE_TYPE_UNSUPPORTED);       //File type unsupported.
550086          return ((ssize_t) -1);
550087        }
550088      //
550089      // Update the file descriptor internal offset.
550090      //
550091      if (size_written > 0)
550092        {
550093          fd->file->offset += size_written;
550094        }
550095      //
550096      // Just return the size written, even if it is an error.
550097      //
550098      return (size_written);
550099  }
```

100.4.13 kernel/fs/file_reference.c

«

Si veda la sezione 99.3.13.

```
560001  #include <kernel/proc.h>
560002  #include <errno.h>
560003  #include <fcntl.h>
560004  //----------------------------------------------------------------------
560005  file_t *
560006  file_reference (int fno)
560007  {
560008      //
```

```
560009      // Check type of request.
560010      //
560011      if (fno < 0)
560012        {
560013          //
560014          // Find a free slot.
560015          //
560016          for (fno = 0; fno < FILE_MAX_SLOTS; fno++)
560017            {
560018              if (file_table[fno].references <= 0)
560019                {
560020                  return (&file_table[fno]);
560021                }
560022            }
560023          return (NULL);
560024        }
560025      else if (fno > FILE_MAX_SLOTS)
560026        {
560027          return (NULL);
560028        }
560029      else
560030        {
560031          return (&file_table[fno]);
560032        }
560033  }
```

100.4.14 kernel/fs/file_stdio_dev_make.c

«

Si veda la sezione 99.3.14.

```
570001  #include <kernel/proc.h>
570002  #include <errno.h>
570003  #include <fcntl.h>
570004  //----------------------------------------------------------------------
570005  file_t *
570006  file_stdio_dev_make (dev_t device, mode_t mode, int oflags)
570007  {
570008      inode_t *inode;
570009      file_t  *file;
570010      //
570011      // Try to allocate a device inode.
570012      //
570013      inode = inode_stdio_dev_make (device, mode);
570014      if (inode == NULL)
570015        {
570016          //
570017          // Variable 'errno' is already set by 'inode_stdio_dev_make()'.
570018          //
```

```
570019          errset (errno);
570020          return (NULL);
570021        }
570022      //
570023      // Inode allocated: need to allocate the system file item.
570024      //
570025      file = file_reference (-1);
570026      if (file == NULL)
570027        {
570028          //
570029          // Remove the inode and return an error.
570030          //
570031          inode_put (inode);
570032          errset (ENFILE);          // Too many files open in system.
570033          return (NULL);
570034        }
570035      //
570036      // Fill with data the system file item.
570037      //
570038      file->references = 1;
570039      file->oflags     = (oflags & (O_RDONLY | O_WRONLY));
570040      file->inode      = inode;
570041      //
570042      // Return system file pointer.
570043      //
570044      return (file);
570045  }
```

100.4.15 kernel/fs/file_table.c

Si veda la sezione 99.3.13.

```
580001  #include <kernel/fs.h>
580002  //----------------------------------------------------------------------
580003  file_t file_table[FILE_MAX_SLOTS];
580004  //----------------------------------------------------------------------
```

100.4.16 kernel/fs/inode_alloc.c

Si veda la sezione 99.3.15.

```
590001  #include <kernel/fs.h>
590002  #include <errno.h>
590003  #include <kernel/k_libc.h>
590004  //----------------------------------------------------------------------
590005  inode_t *
590006  inode_alloc (dev_t device, mode_t mode, uid_t uid)
```

```
590007 {
590008     sb_t          *sb;
590009     inode_t       *inode;
590010     int            m;                    // Index inside the inode map.
590011     int            map_element;
590012     int            map_bit;
590013     int            map_mask;
590014     ino_t          ino;
590015     //
590016     // Check for arguments.
590017     //
590018     if (mode == 0)
590019       {
590020         errset (EINVAL);         // Invalid argument.
590021         return (NULL);
590022       }
590023     //
590024     // Get the super block from the known device.
590025     //
590026     sb = sb_reference (device);
590027     if (sb == NULL)
590028       {
590029         errset (ENODEV);         // No such device.
590030         return (NULL);
590031       }
590032     //
590033     // Find a free inode.
590034     //
590035     while (1)
590036       {
590037         //
590038         // Scan the inode bit map, to find a free inode
590039         // for new allocation.
590040         //
590041         for (m = 0; m < (SB_MAP_INODE_SIZE * 16); m++)
590042           {
590043             map_element = m / 16;
590044             map_bit     = m % 16;
590045             map_mask    = 1 << map_bit;
590046             if (!(sb->map_inode[map_element] & map_mask))
590047               {
590048                 //
590049                 // Found a free element: change the map to
590050                 // allocate the inode.
590051                 //
590052                 sb->map_inode[map_element] |= map_mask;
590053                 sb->changed = 1;
590054                 ino = m;  // Found a free inode:
```

```
590055                  break;           // exit the scan loop.
590056                }
590057            }
590058        //
590059        // Check if the scan was successful.
590060        //
590061        if (ino == 0)
590062          {
590063            errset (ENOSPC);    // No space left on device.
590064            return (NULL);
590065          }
590066        //
590067        // The inode was allocated inside the map in memory.
590068        //
590069        inode = inode_get (device, ino);
590070        if (inode == NULL)
590071          {
590072            errset (ENFILE);    // Too many files open in system.
590073            return (NULL);
590074          }
590075        //
590076        // Verify if the inode is really free: if it isn't, must save
590077        // it to disk.
590078        //
590079        if (inode->size > 0 || inode->links > 0)
590080          {
590081            //
590082            // Strange: should not have a size! Check if there are even
590083            // links. Please note that 255 links (that is -1) is to be
590084            // considered a free inode, marked in a special way for some
590085            // unknown reason. Currently, 'LINK_MAX' is equal to 254,
590086            // for that reason.
590087            //
590088            if (inode->links > 0 && inode->links < LINK_MAX)
590089              {
590090                //
590091                // Tell something.
590092                //
590093                k_printf ("kernel alert: device %04x: "
590094                          "found \"free\" inode %i "
590095                          "that still has size %i "
590096                          "and %i links!\n",
590097                          device, ino, inode->size, inode->links);
590098                //
590099                // The inode must be set again to free, inside
590100                // the bit map.
590101                //
590102                map_element = ino / 16;
```

```
590103                     map_bit      = ino % 16;
590104                     map_mask     = 1 << map_bit;
590105                     sb->map_inode[map_element] &= ~map_mask;
590106                     sb->changed = 1;
590107                     //
590108                     // Try to fix: reset all to zero.
590109                     //
590110                     inode->mode      = 0;
590111                     inode->uid       = 0;
590112                     inode->gid       = 0;
590113                     inode->time      = 0;
590114                     inode->links     = 0;
590115                     inode->size      = 0;
590116                     inode->direct[0] = 0;
590117                     inode->direct[1] = 0;
590118                     inode->direct[2] = 0;
590119                     inode->direct[3] = 0;
590120                     inode->direct[4] = 0;
590121                     inode->direct[5] = 0;
590122                     inode->direct[6] = 0;
590123                     inode->indirect1 = 0;
590124                     inode->indirect2 = 0;
590125                     inode->changed   = 1;
590126                     //
590127                     // Save fixed inode to disk.
590128                     //
590129                     inode_put (inode);
590130                     continue;
590131                   }
590132                 else
590133                   {
590134                     //
590135                     // Truncate the inode, save and break.
590136                     //
590137                     inode_truncate (inode);
590138                     inode_save (inode);
590139                     break;
590140                   }
590141               }
590142             else
590143               {
590144                 //
590145                 // Considering free the inode found.
590146                 //
590147                 break;
590148               }
590149           }
590150         //
```

```
590151      // Put data inside the inode.
590152      //
590153      inode->mode    = mode;
590154      inode->uid     = uid;
590155      inode->gid     = 0;
590156      inode->size    = 0;
590157      inode->time    = k_time (NULL);
590158      inode->links   = 0;
590159      inode->changed = 1;
590160      //
590161      // Save the inode.
590162      //
590163      inode_save (inode);
590164      //
590165      // Return the inode pointer.
590166      //
590167      return (inode);
590168  }
```

100.4.17 kernel/fs/inode_check.c

«

Si veda la sezione 99.3.16.

```
600001  #include <kernel/fs.h>
600002  #include <errno.h>
600003  #include <kernel/k_libc.h>
600004  //----------------------------------------------------------------------
600005  int
600006  inode_check (inode_t *inode, mode_t type, int perm, uid_t uid)
600007  {
600008      //
600009      // Ensure that the variable 'type' has only the requested file type.
600010      //
600011      type = (type & S_IFMT);
600012      //
600013      // Check inode argument.
600014      //
600015      if (inode == NULL)
600016        {
600017          errset (EINVAL);        // Invalid argument.
600018          return (-1);
600019        }
600020      //
600021      // The inode is not NULL: verify that the inode is of a type
600022      // allowed (the parameter 'type' can hold more than one
600023      // possibility).
600024      //
600025      if (!(inode->mode & type))
```

```
600026          {
600027            errset (E_FILE_TYPE);    // The file type is not
600028            return (-1);             // the expected one.
600029          }
600030        //
600031        // The file type is correct.
600032        //
600033        if (inode->uid != 0 && uid == 0)
600034          {
600035            return (0);              // The root user has all permissions.
600036          }
600037        //
600038        // The user is not root or the inode is owned by root.
600039        //
600040        if (inode->uid == uid)
600041          {
600042            //
600043            // The user own the inode and must check user permissions.
600044            //
600045            perm = (perm << 6);
600046            if ((inode->mode & perm) ^ perm)
600047              {
600048                errset (EACCES);             // Permission denied.
600049                return (-1);
600050              }
600051            else
600052              {
600053                return (0);
600054              }
600055          }
600056        //
600057        // The user does not own the inode: the other permissions are
600058        // checked.
600059        //
600060        if ((inode->mode & perm) ^ perm)
600061          {
600062            errset (EACCES);                 // Permission denied.
600063            return (-1);
600064          }
600065        else
600066          {
600067            return (0);
600068          }
600069    }
```

100.4.18 kernel/fs/inode_dir_empty.c

«

Si veda la sezione 99.3.17.

```
610001 #include <kernel/fs.h>
610002 #include <errno.h>
610003 #include <kernel/k_libc.h>
610004 //----------------------------------------------------------------------
610005 int
610006 inode_dir_empty (inode_t *inode)
610007 {
610008     off_t        start;
610009     char         buffer[SB_MAX_ZONE_SIZE];
610010     directory_t *dir;
610011     ssize_t      size_read;
610012     int          d;                       // Directory buffer index.
610013     //
610014     // Check argument: must be a directory.
610015     //
610016     if (inode == NULL || !S_ISDIR (inode->mode))
610017       {
610018         errset (EINVAL);                  // Invalid argument.
610019         return (0);                       // false
610020       }
610021     //
610022     // Read the directory content: if an item is present (except '.' and
610023     // '..'), the directory is not empty.
610024     //
610025     for (start  = 0;
610026          start <  inode->size;
610027          start += inode->sb->blksize)
610028       {
610029         size_read = inode_file_read (inode, start, buffer,
610030                                      inode->sb->blksize,
610031                                      NULL);
610032         if (size_read < sizeof (directory_t))
610033           {
610034             break;
610035           }
610036         //
610037         // Scan the directory portion just read.
610038         //
610039         dir = (directory_t *) buffer;
610040         //
610041         for (d = 0; d < size_read; d += (sizeof (directory_t)), dir++)
610042           {
610043             if (dir->ino != 0                                 &&
610044                 strncmp (dir->name, ".", NAME_MAX) != 0     &&
610045                 strncmp (dir->name, "..", NAME_MAX) != 0)
```

```
610046                {
610047                  //
610048                  // There is an item and the directory is not empty.
610049                  //
610050                  return (0);     // false
610051                }
610052            }
610053        }
610054      //
610055      // Nothing was found; good!
610056      //
610057      return (1);         // true
610058  }
```

100.4.19 kernel/fs/inode_file_read.c

«

Si veda la sezione 99.3.18.

```
620001  #include <kernel/fs.h>
620002  #include <errno.h>
620003  #include <kernel/k_libc.h>
620004  //----------------------------------------------------------------------
620005  ssize_t
620006  inode_file_read (inode_t *inode, off_t offset,
620007                   void *buffer, size_t count, int *eof)
620008  {
620009      unsigned char *destination = (unsigned char *) buffer;
620010      unsigned char  zone_buffer[SB_MAX_ZONE_SIZE];
620011      blkcnt_t       blkcnt_read;
620012      off_t          off_fzone;   // File zone offset.
620013      off_t          off_buffer;  // Destination buffer offset.
620014      ssize_t        size_read;   // Byte transfer counter.
620015      zno_t          fzone;
620016      off_t          off_end;
620017      //
620018      // The inode pointer must be valid, and
620019      // the start byte must be positive.
620020      //
620021      if (inode == NULL || offset < 0)
620022        {
620023          errset (EINVAL);        // Invalid argument.
620024          return ((ssize_t) -1);
620025        }
620026      //
620027      // Check if the start address is inside the file size. This is not
620028      // an error, but zero bytes are read and `*eof' is set. Otherwise,
620029      // `*eof' is reset.
620030      //
```

```
620031      if (offset >= inode->size)
620032        {
620033          (eof != NULL)? *eof = 1: 0;
620034          return (0);
620035        }
620036      else
620037        {
620038          (eof != NULL)? *eof = 0: 0;
620039        }
620040      //
620041      // Adjust, if necessary, the size of read, because it cannot be
620042      // larger than the actual file size. The variable 'off_end' is
620043      // used to calculate the position *after* the requested read.
620044      // Remember that the first file position is byte zero; so,
620045      // the byte index inside the file goes from zero to inode->size -1.
620046      //
620047      off_end  = offset;
620048      off_end += count;
620049      if (off_end > inode->size)
620050        {
620051          count = (inode->size - offset);
620052        }
620053      //
620054      // Read the first file-zone inside the zone buffer.
620055      //
620056      fzone       = offset / inode->sb->blksize;
620057      off_fzone   = offset % inode->sb->blksize;
620058      blkcnt_read = inode_fzones_read (inode, fzone, zone_buffer,
620059                                       (blkcnt_t) 1);
620060      if (blkcnt_read <= 0)
620061        {
620062          //
620063          // Sorry!
620064          //
620065          return (0);                // Zero bytes read!
620066        }
620067      //
620068      // The first file-zone was read: copy it inside the destination
620069      // buffer and continue reading the other zones needed. Variables
620070      // 'off_buffer' (destination buffer index) and 'size_read' (copy
620071      // byte counter) must be reset here. Variable 'off_fzone' is already
620072      // set with the initial offset inside 'zone_buffer'.
620073      //
620074      off_buffer = 0;
620075      size_read  = 0;
620076      //
620077      while (count)
620078        {
```

```
620079            //
620080            // Copy the zone buffer into the destination. Variables
620081            // `off_fzone', `off_buffer' and `size_read' must not be
620082            // initialized inside the loop.
620083            //
620084            for (; off_fzone < inode->sb->blksize && count > 0;
620085                 off_fzone++, off_buffer++, size_read++,
620086                 count--, offset++)
620087              {
620088                destination[off_buffer] = zone_buffer[off_fzone];
620089              }
620090            //
620091            // If not all the bytes are copied, read the next file-zone.
620092            //
620093            if (count)
620094              {
620095                //
620096                // Read another file-zone inside the zone buffer.
620097                // Again, the function `inode_fzones_read()' might
620098                // return a null pointer, but the variable `errno' tells if
620099                // it is really an error. For this reason, the variable
620100                // `errno' must be reset before the read, and checked after
620101                // it.
620102                //
620103                fzone       = offset / inode->sb->blksize;
620104                off_fzone   = offset % inode->sb->blksize;
620105                blkcnt_read = inode_fzones_read (inode, fzone, zone_buffer,
620106                                                 (blkcnt_t) 1);
620107                if (blkcnt_read <= 0)
620108                  {
620109                    //
620110                    // Sorry: only `size_read' bytes read!
620111                    //
620112                    return (size_read);
620113                  }
620114              }
620115          }
620116        //
620117        // The requested size was read completely.
620118        //
620119        return (size_read);
620120    }
```

100.4.20 kernel/fs/inode_file_write.c

«

Si veda la sezione 99.3.19.

```
630001 #include <kernel/fs.h>
630002 #include <errno.h>
630003 #include <kernel/k_libc.h>
630004 //----------------------------------------------------------------------
630005 ssize_t
630006 inode_file_write (inode_t *inode, off_t offset, void *buffer,
630007                   size_t count)
630008 {
630009     unsigned char *buffer_source = (unsigned char *) buffer;
630010     unsigned char  buffer_zone[SB_MAX_ZONE_SIZE];
630011     off_t          off_fzone;    // File zone offset.
630012     off_t          off_source;   // Source buffer offset.
630013     ssize_t        size_copied;  // Byte transfer counter.
630014     ssize_t        size_written; // Byte written counter.
630015     zno_t          fzone;
630016     zno_t          zone;
630017     blkcnt_t       blkcnt_read;
630018     int            status;
630019     //
630020     // The inode pointer must be valid, and
630021     // the start byte must be positive.
630022     //
630023     if (inode == NULL || offset < 0)
630024       {
630025         errset (EINVAL);         // Invalid argument.
630026         return ((ssize_t) -1);
630027       }
630028     //
630029     // Read a zone, modify it with the source buffer, then write it back
630030     // and continue reading and writing other zones if needed.
630031     //
630032     for (size_written = 0, off_source = 0, size_copied = 0;
630033          count > 0; size_written += size_copied)
630034       {
630035         //
630036         // Read the next file-zone inside the zone buffer: the function
630037         // `inode_zone()' is used to create automatically the zone, if
630038         // it does not exist.
630039         //
630040         fzone     = offset / inode->sb->blksize;
630041         off_fzone = offset % inode->sb->blksize;
630042         zone      = inode_zone (inode, fzone, 1);
630043         if (zone == 0)
630044           {
630045             //
```

```
630046                // Return previously written bytes. The variable 'errno' is
630047                // already set by 'inode_zone()'.
630048                //
630049                return (size_written);
630050              }
630051            blkcnt_read = inode_fzones_read (inode, fzone, buffer_zone,
630052                                             (blkcnt_t) 1);
630053            if (blkcnt_read <= 0)
630054              {
630055                //
630056                // Even if the value is zero, there is a problem reading the
630057                // zone to be overwritten (because 'inode_zone()' should
630058                // have already created such zone). The variable 'errno' is
630059                // already set by 'inode_fzones_read()'.
630060                //
630061                return ((ssize_t) -1);
630062              }
630063            //
630064            // The zone was successfully loaded inside the buffer: overwrite
630065            // the zone buffer with the source buffer.
630066            //
630067            for (size_copied = 0;
630068                 off_fzone < inode->sb->blksize && count > 0;
630069                 off_fzone++, off_source++, size_copied++, count--,
630070                 offset++)
630071              {
630072                buffer_zone[off_fzone] = buffer_source[off_source];
630073              }
630074            //
630075            // Save the zone.
630076            //
630077            status = zone_write (inode->sb, zone, buffer_zone);
630078            if (status != 0)
630079              {
630080                //
630081                // Cannot save the zone: return the size already written.
630082                // The variable 'errno' is already set by 'zone_write()'.
630083                //
630084                return (size_written);
630085              }
630086            //
630087            // Zone saved: update the file size if necessary (and the inode
630088            // too).
630089            //
630090            if (inode->size <= offset)
630091              {
630092                inode->size    = offset;
630093                inode->changed = 1;
```

```
630094                inode_save (inode);
630095              }
630096          }
630097        //
630098        // All done successfully: return the value.
630099        //
630100        return (size_written);
630101  }
```

100.4.21 kernel/fs/inode_free.c

«

Si veda la sezione 99.3.20.

```
640001  #include <kernel/fs.h>
640002  #include <errno.h>
640003  #include <kernel/k_libc.h>
640004  //----------------------------------------------------------------------
640005  int
640006  inode_free (inode_t *inode)
640007  {
640008      int      map_element;
640009      int      map_bit;
640010      int      map_mask;
640011      //
640012      if (inode == NULL)
640013        {
640014          errset (EINVAL);         // Invalid argument.
640015          return (-1);
640016        }
640017      //
640018      map_element = inode->ino / 16;
640019      map_bit     = inode->ino % 16;
640020      map_mask    = 1 << map_bit;
640021      //
640022      if (inode->sb->map_inode[map_element] & map_mask)
640023        {
640024          inode->sb->map_inode[map_element] -= map_mask;
640025          inode->sb->changed = 1;
640026        }
640027      //
640028      inode->mode       = 0;
640029      inode->uid        = 0;
640030      inode->gid        = 0;
640031      inode->size       = 0;
640032      inode->time       = 0;
640033      inode->links      = 0;
640034      inode->changed    = 1;
640035      inode->references = 0;
```

```
640036     //
640037     return (inode_save (inode));
640038 }
```

100.4.22 kernel/fs/inode_fzones_read.c

«

Si veda la sezione 99.3.21.

```
650001 #include <kernel/fs.h>
650002 #include <errno.h>
650003 #include <kernel/k_libc.h>
650004 //----------------------------------------------------------------------
650005 blkcnt_t
650006 inode_fzones_read (inode_t *inode, zno_t zone_start,
650007                    void *buffer, blkcnt_t blkcnt)
650008 {
650009     unsigned char *destination = (unsigned char *) buffer;
650010     int            status;       // 'zone_read()' return value.
650011     blkcnt_t       blkcnt_read;  // Zone counter/index.
650012     zno_t          zone;
650013     zno_t          fzone;
650014     //
650015     // Read the zones into the destination buffer.
650016     //
650017     for (blkcnt_read = 0, fzone = zone_start;
650018          blkcnt_read < blkcnt;
650019          blkcnt_read++, fzone++)
650020       {
650021         //
650022         // Calculate the zone number, from the file-zone, reading the
650023         // inode. If a zone is not really allocated, the result is zero
650024         // and is valid.
650025         //
650026         zone = inode_zone (inode, fzone, 0);
650027         if (zone == ((zno_t) -1))
650028           {
650029             //
650030             // This is an error. Return the read zones quantity.
650031             //
650032             return (blkcnt_read);
650033           }
650034         //
650035         // Update the destination buffer pointer.
650036         //
650037         destination += (blkcnt_read * inode->sb->blksize);
650038         //
650039         // Read the zone inside the destination buffer, but if the zone
650040         // is zero, a zeroed zone must be filled.
```

```
650041          //
650042          if (zone == 0)
650043            {
650044              memset (destination, 0, (size_t) inode->sb->blksize);
650045            }
650046          else
650047            {
650048              status = zone_read (inode->sb, zone, destination);
650049              if (status != 0)
650050                {
650051                  //
650052                  // Could not read the requested zone: return the zones
650053                  // read correctly.
650054                  //
650055                  errset (EIO);                  // I/O error.
650056                  return (blkcnt_read);
650057                }
650058            }
650059        }
650060      //
650061      // All zones read correctly inside the buffer.
650062      //
650063      return (blkcnt_read);
650064  }
```

100.4.23 kernel/fs/inode_fzones_write.c

«

Si veda la sezione 99.3.21.

```
660001  #include <kernel/fs.h>
660002  #include <errno.h>
660003  #include <kernel/k_libc.h>
660004  //------------------------------------------------------------------------
660005  blkcnt_t
660006  inode_fzones_write (inode_t *inode, zno_t zone_start, void *buffer,
660007                      blkcnt_t blkcnt)
660008  {
660009      unsigned char *source = (unsigned char *) buffer;
660010      int            status;              // 'zone_read()' return value.
660011      blkcnt_t       blkcnt_written;      // Written zones counter.
660012      zno_t          zone;
660013      zno_t          fzone;
660014      //
660015      // Write the zones into the destination buffer.
660016      //
660017      for (blkcnt_written = 0, fzone = zone_start;
660018           blkcnt_written < blkcnt;
660019           blkcnt_written++, fzone++)
```

```
660020          {
660021            //
660022            // Find real zone from file-zone.
660023            //
660024            zone = inode_zone (inode, fzone, 1);
660025            if (zone == 0 || zone == ((zno_t) -1))
660026              {
660027                //
660028                // Function 'inode_zone()' should allocate automatically
660029                // a missing zone and should return a valid zone or
660030                // (zno_t) -1. Anyway, even if a zero zone is returned,
660031                // it is an error. Return the 'blkcnt_written' value.
660032                //
660033                return (blkcnt_written);
660034              }
660035            //
660036            // Update the source buffer pointer for the next zone write.
660037            //
660038            source += (blkcnt_written * inode->sb->blksize);
660039            //
660040            // Write the zone from the buffer content.
660041            //
660042            status  = zone_write (inode->sb, zone, source);
660043            if (status != 0)
660044              {
660045                //
660046                // Cannot write the zone. Return 'size_written_zone' value.
660047                //
660048                return (blkcnt_written);
660049              }
660050          }
660051        //
660052        // All zones read correctly inside the buffer.
660053        //
660054        return (blkcnt_written);
660055  }
```

100.4.24 kernel/fs/inode_get.c

«

Si veda la sezione 99.3.23.

```
670001  #include <kernel/fs.h>
670002  #include <errno.h>
670003  #include <kernel/k_libc.h>
670004  #include <kernel/devices.h>
670005  //----------------------------------------------------------------------
670006  inode_t *
670007  inode_get (dev_t device, ino_t ino)
```

```
670008 {
670009     sb_t              *sb;
670010     inode_t           *inode;
670011     unsigned long int start;
670012     size_t             size;
670013     ssize_t            n;
670014     int                status;
670015     //
670016     // Verify if the root file system inode was requested.
670017     //
670018     if (device == 0 && ino == 1)
670019       {
670020         //
670021         // Get root file system inode.
670022         //
670023         inode = inode_reference (device, ino);
670024         if (inode == NULL)
670025           {
670026             //
670027             // The file system root directory inode is not yet loaded:
670028             // get the first super block.
670029             //
670030             sb = sb_reference ((dev_t) 0);
670031             if (sb == NULL || sb->device == 0)
670032               {
670033                 //
670034                 // This error should never happen.
670035                 //
670036                 errset (EUNKNOWN);        // Unknown error.
670037                 return (NULL);
670038               }
670039             //
670040             // Load the file system root directory inode (recursive
670041             // call).
670042             //
670043             inode = inode_get (sb->device, (ino_t) 1);
670044             if (inode == NULL)
670045               {
670046                 //
670047                 // This error should never happen.
670048                 //
670049                 return (NULL);
670050               }
670051             //
670052             // Return the directory inode.
670053             //
670054             return (inode);
670055           }
```

```
670056          else
670057            {
670058              //
670059              // The file system root directory inode is already
670060              // available.
670061              //
670062              if (inode->references >= INODE_MAX_REFERENCES)
670063                {
670064                  errset (ENFILE); // Too many files open in system.
670065                  return (NULL);
670066                }
670067              else
670068                {
670069                  inode->references++;
670070                  return (inode);
670071                }
670072            }
670073        }
670074      //
670075      // A common device-inode pair was requested: try to find an already
670076      // cached inode.
670077      //
670078      inode = inode_reference (device, ino);
670079      if (inode != NULL)
670080        {
670081          if (inode->references >= INODE_MAX_REFERENCES)
670082            {
670083              errset (ENFILE);    // Too many files open in system.
670084              return (NULL);
670085            }
670086          else
670087            {
670088              inode->references++;
670089              return (inode);
670090            }
670091        }
670092      //
670093      // The inode is not yet available: get super block.
670094      //
670095      sb = sb_reference (device);
670096      if (sb == NULL)
670097        {
670098          errset (ENODEV);        // No such device.
670099          return (NULL);
670100        }
670101      //
670102      // The super block is available, but the inode is not yet cached.
670103      // Verify if the inode map reports it as allocated.
```

```
670104      //
670105      status = sb_inode_status (sb, ino);
670106      if (!status)
670107        {
670108          //
670109          // The inode is not allocated and cannot be loaded.
670110          //
670111          errset (ENOENT);        // No such file or directory.
670112          return (NULL);
670113        }
670114      //
670115      // The inode was not already cached, but is considered as allocated
670116      // inside the inode map. Find a free slot to load the inode inside
670117      // the inode table (in memory).
670118      //
670119      inode = inode_reference ((dev_t) -1, (ino_t) -1);
670120      if (inode == NULL)
670121        {
670122          errset (ENFILE);        // Too many files open in system.
670123          return (NULL);
670124        }
670125      //
670126      // A free inode slot was found. The inode must be loaded.
670127      // Calculate the memory inode size, to be saved inside the file
670128      // system: the administrative inode data, as it is saved inside
670129      // the file system. The 'inode_t' type is bigger than the real
670130      // inode administrative size, because it contains more data, that is
670131      // not saved on disk.
670132      //
670133      size   = offsetof (inode_t, sb);
670134      //
670135      // Calculating start position for read.
670136      //
670137      // [1] Boot block.
670138      // [2] Super block.
670139      // [3] Inode bit map.
670140      // [4] Zone bit map.
670141      // [5] Previous inodes: consider that the inode zero is
670142      //     present in the inode map, but not in the inode
670143      //     table.
670144      //
670145      start  = 1024;                              // [1]
670146      start += 1024;                              // [2]
670147      start += (sb->map_inode_blocks * 1024);   // [3]
670148      start += (sb->map_zone_blocks * 1024);    // [4]
670149      start += ((ino -1) * size);               // [5]
670150      //
670151      // Read inode from disk.
```

```
670152      //
670153      n = dev_io ((pid_t) -1, device, DEV_READ, start, inode, size, NULL);
670154      if (n != size)
670155        {
670156          errset (EIO);                   // I/O error.
670157          return (NULL);
670158        }
670159      //
670160      // The inode was read: add some data to the working copy in memory.
670161      //
670162      inode->sb            = sb;
670163      inode->sb_attached   = NULL;
670164      inode->ino           = ino;
670165      inode->references    = 1;
670166      inode->changed       = 0;
670167      //
670168      inode->blkcnt        = inode->size;
670169      inode->blkcnt       /= sb->blksize;
670170      if (inode->size % sb->blksize)
670171        {
670172          inode->blkcnt++;
670173        }
670174      //
670175      // Return the inode pointer.
670176      //
670177      return (inode);
670178  }
```

100.4.25 kernel/fs/inode_put.c

Si veda la sezione 99.3.24.

```
680001  #include <kernel/fs.h>
680002  #include <errno.h>
680003  #include <kernel/k_libc.h>
680004  //----------------------------------------------------------------------
680005  int
680006  inode_put (inode_t *inode)
680007  {
680008      int         status;
680009      //
680010      // Check for valid argument.
680011      //
680012      if (inode == NULL)
680013        {
680014          errset (EINVAL);                // Invalid argument.
680015          return (-1);
680016        }
```

```
680017      //
680018      // Check for valid references.
680019      //
680020      if (inode->references <= 0)
680021        {
680022          errset (EUNKNOWN);              // Cannot put an inode with
680023          return (-1);                    // zero or negative references.
680024        }
680025      //
680026      // Debug.
680027      //
680028      if (inode->sb->device == 0 && inode->ino != 0)
680029        {
680030          k_printf ("kernel alert: trying to close inode with device "
680031                    "zero, but a number different than zero!\n");
680032          errset (EUNKNOWN);              // Cannot put an inode with
680033          return (-1);                    // zero or negative references.
680034        }
680035      //
680036      // There is at least one reference: now the references value is
680037      // reduced.
680038      //
680039      inode->references--;
680040      inode->changed = 1;
680041      //
680042      // If 'inode->ino' is zero, it means that the inode was created in
680043      // memory, but there is no file system for it. For example, it might
680044      // be a standard I/O inode create automatically for a process.
680045      // Inodes with number zero cannot be removed from a file system.
680046      //
680047      if (inode->ino == 0)
680048        {
680049          //
680050          // Nothing to do: just return.
680051          //
680052          return (0);
680053        }
680054      //
680055      //  References counter might be zero.
680056      //
680057      if (inode->references == 0)
680058        {
680059          //
680060          // Check if the inode is to be deleted (until there are
680061          // run time references, the inode cannot be removed).
680062          //
680063          if (inode->links == 0
680064           || (S_ISDIR (inode->mode) && inode->links == 1))
```

```
680065              {
680066                //
680067                // The inode has no more run time references and file system
680068                // links are also zero (or one for a directory): remove it!
680069                //
680070                status = inode_truncate (inode);
680071                if (status != 0)
680072                  {
680073                    k_perror (NULL);
680074                  }
680075                //
680076                inode_free (inode);
680077                return (0);
680078              }
680079          }
680080        //
680081        // Save inode to disk and return.
680082        //
680083        return (inode_save (inode));
680084  }
```

100.4.26 kernel/fs/inode_reference.c

Si veda la sezione 99.3.25.

```
690001  #include <kernel/fs.h>
690002  #include <errno.h>
690003  #include <kernel/k_libc.h>
690004  //----------------------------------------------------------------------
690005  inode_t *
690006  inode_reference (dev_t device, ino_t ino)
690007  {
690008      int    s;                              // Slot index.
690009      sb_t *sb_table = sb_reference (0);
690010      //
690011      // If device is zero, and inode is zero, a reference to the whole
690012      // table is returned.
690013      //
690014      if (device == 0 && ino == 0)
690015        {
690016          return (inode_table);
690017        }
690018      //
690019      // If device is ((dev_t) -1) and the inode is ((ino_t) -1), a
690020      // reference to a free inode slot is returned.
690021      //
690022      if (device == (dev_t) -1 && ino == ((ino_t) -1))
690023        {
```

```
690024          for (s = 0; s < INODE_MAX_SLOTS; s++)
690025            {
690026              if (inode_table[s].references == 0)
690027                {
690028                  return (&inode_table[s]);
690029                }
690030            }
690031          return (NULL);
690032        }
690033      //
690034      // If device is zero and the inode is 1, a reference to the root
690035      // directory inode is returned.
690036      //
690037      if (device == 0 && ino == 1)
690038        {
690039          //
690040          // The super block table is to be scanned.
690041          //
690042          for (device = 0, s = 0; s < SB_MAX_SLOTS; s++)
690043            {
690044              if (sb_table[s].device != 0                  &&
690045                  sb_table[s].inode_mounted_on == NULL)
690046                {
690047                  device = sb_table[s].device;
690048                  break;
690049                }
690050            }
690051          if (device == 0)
690052            {
690053              errset (E_CANNOT_FIND_ROOT_DEVICE);
690054              return (NULL);
690055            }
690056          //
690057          // Scan the inode table to find inode 1 and the same device.
690058          //
690059          for (s = 0; s < INODE_MAX_SLOTS; s++)
690060            {
690061              if (inode_table[s].sb->device == device     &&
690062                  inode_table[s].ino == 1)
690063                {
690064                  return (&inode_table[s]);
690065                }
690066            }
690067          //
690068          // Cannot find a root file system inode.
690069          //
690070          errset (E_CANNOT_FIND_ROOT_INODE);
690071          return (NULL);
```

```
690072          }
690073        //
690074        // A device and an inode number were selected: find the inode
690075        // associated to it.
690076        //
690077        for (s = 0; s < INODE_MAX_SLOTS; s++)
690078          {
690079            if (inode_table[s].sb->device == device &&
690080                inode_table[s].ino == ino)
690081              {
690082                return (&inode_table[s]);
690083              }
690084          }
690085        //
690086        // The inode was not found.
690087        //
690088        return (NULL);
690089    }
```

100.4.27 kernel/fs/inode_save.c

«

Si veda la sezione 99.3.26.

```
700001  #include <kernel/fs.h>
700002  #include <errno.h>
700003  #include <kernel/k_libc.h>
700004  #include <kernel/devices.h>
700005  //----------------------------------------------------------------------
700006  int
700007  inode_save (inode_t *inode)
700008  {
700009      size_t            size;
700010      unsigned long int start;
700011      ssize_t           n;
700012      //
700013      // Check for valid argument.
700014      //
700015      if (inode == NULL)
700016        {
700017          errset (EINVAL);                 // Invalid argument.
700018          return (-1);
700019        }
700020      //
700021      // If the inode number is zero, no file system is involved!
700022      //
700023      if (inode->ino == 0)
700024        {
700025          return (0);
```

```
700026         }
700027       //
700028       // Save the super block to disk.
700029       //
700030       sb_save (inode->sb);
700031       //
700032       // Save the inode to disk.
700033       //
700034       if (inode->changed)
700035         {
700036           size   = offsetof (inode_t, sb);
700037           //
700038           // Calculating start position for write.
700039           //
700040           // [1] Boot block.
700041           // [2] Super block.
700042           // [3] Inode bit map.
700043           // [4] Zone bit map.
700044           // [5] Previous inodes: consider that the inode zero is
700045           //     present in the inode map, but not in the inode
700046           //     table.
700047           //
700048           start  = 1024;                                 // [1]
700049           start += 1024;                                 // [2]
700050           start += (inode->sb->map_inode_blocks * 1024);  // [3]
700051           start += (inode->sb->map_zone_blocks * 1024);   // [4]
700052           start += ((inode->ino -1) * size);              // [5]
700053           //
700054           // Write the inode.
700055           //
700056           n = dev_io ((pid_t) -1, inode->sb->device, DEV_WRITE, start,
700057                       inode, size, NULL);
700058           //
700059           inode->changed = 0;
700060         }
700061       return (0);
700062   }
```

100.4.28 kernel/fs/inode_stdio_dev_make.c

«

Si veda la sezione 99.3.27.

```
710001 #include <kernel/fs.h>
710002 #include <errno.h>
710003 #include <kernel/k_libc.h>
710004 //----------------------------------------------------------------------
710005 inode_t *
710006 inode_stdio_dev_make (dev_t device, mode_t mode)
```

```
710007 {
710008     inode_t      *inode;
710009     //
710010     // Check for arguments.
710011     //
710012     if (mode == 0 || device == 0)
710013       {
710014         errset (EINVAL);         // Invalid argument.
710015         return (NULL);
710016       }
710017     //
710018     // Find a free inode.
710019     //
710020     inode = inode_reference ((dev_t) -1, (ino_t) -1);
710021     if (inode == NULL)
710022       {
710023         //
710024         // No free slot available.
710025         //
710026         errset (ENFILE);         // Too many files open in system.
710027         return (NULL);
710028       }
710029     //
710030     // Put data inside the inode. Please note that 'inode->ino' must be
710031     // zero, because it is necessary to recognize it as an internal
710032     // inode with no file system. Otherwise, with a value different than
710033     // zero, 'inode_put()' will try to remove it. [*]
710034     //
710035     inode->mode          = mode;
710036     inode->uid           = 0;
710037     inode->gid           = 0;
710038     inode->size          = 0;
710039     inode->time          = k_time (NULL);
710040     inode->links         = 0;
710041     inode->direct[0]     = device;
710042     inode->direct[1]     = 0;
710043     inode->direct[2]     = 0;
710044     inode->direct[3]     = 0;
710045     inode->direct[4]     = 0;
710046     inode->direct[5]     = 0;
710047     inode->direct[6]     = 0;
710048     inode->indirect1     = 0;
710049     inode->indirect2     = 0;
710050     inode->sb_attached   = NULL;
710051     inode->sb            = 0;
710052     inode->ino           = 0;    // Must be zero. [*]
710053     inode->blkcnt        = 0;
710054     inode->references    = 1;
```

```
710055      inode->changed        = 0;
710056      //
710057      // Add all access permissions.
710058      //
710059      inode->mode          |= (S_IRWXU|S_IRWXG|S_IRWXO);
710060      //
710061      // Return the inode pointer.
710062      //
710063      return (inode);
710064  }
```

100.4.29 kernel/fs/inode_table.c

Si veda la sezione 99.3.25.

```
720001  #include <kernel/fs.h>
720002  //----------------------------------------------------------------------
720003  inode_t inode_table[INODE_MAX_SLOTS];
```

100.4.30 kernel/fs/inode_truncate.c

Si veda la sezione 99.3.28.

```
730001  #include <kernel/fs.h>
730002  #include <errno.h>
730003  #include <kernel/k_libc.h>
730004  //----------------------------------------------------------------------
730005  int
730006  inode_truncate (inode_t *inode)
730007  {
730008    unsigned int indirect_zones;
730009    zno_t        zone_table1[INODE_MAX_INDIRECT_ZONES];
730010    zno_t        zone_table2[INODE_MAX_INDIRECT_ZONES];
730011    unsigned int i;              // Direct index.
730012    unsigned int i0;             // Single indirect index.
730013    unsigned int i1;             // Double indirect first index.
730014    unsigned int i2;             // Double indirect second index.
730015    int          status;         // 'zone_read()' return value.
730016    //
730017    // Calculate how many indirect zone numbers are stored inside
730018    // a zone: it depends on the zone size.
730019    //
730020    indirect_zones   = inode->sb->blksize / 2;
730021    //
730022    // Scan and release direct zones. Errors are ignored.
730023    //
730024    for (i = 0; i < 7; i++)
```

```
730025        {
730026          zone_free (inode->sb, inode->direct[i]);
730027          inode->direct[i] = 0;
730028        }
730029      //
730030      // Scan single indirect zones, if present.
730031      //
730032      if (inode->blkcnt > 7 && inode->indirect1 != 0)
730033        {
730034          //
730035          // There is a single indirect table to load. Errors are
730036          // almost ignored.
730037          //
730038          status = zone_read (inode->sb, inode->indirect1, zone_table1);
730039          if (status == 0)
730040            {
730041              //
730042              // Scan the table and remove zones.
730043              //
730044              for (i0 = 0; i0 < indirect_zones; i0++)
730045                {
730046                  zone_free (inode->sb, zone_table1[i0]);
730047                }
730048            }
730049          //
730050          // Remove indirect table too.
730051          //
730052          zone_free (inode->sb, inode->indirect1);
730053          //
730054          // Clear single indirect reference inside the inode.
730055          //
730056          inode->indirect1 = 0;
730057        }
730058      //
730059      // Scan double indirect zones, if present.
730060      //
730061      if (   inode->blkcnt > (7+indirect_zones)
730062          && inode->indirect2 != 0)
730063        {
730064          //
730065          // There is a double indirect table to load. Errors are
730066          // almost ignored.
730067          //
730068          status = zone_read (inode->sb, inode->indirect2, zone_table1);
730069          if (status == 0)
730070            {
730071              //
730072              // Scan the table and get second level indirection.
```

```
730073            //
730074            for (i1 = 0; i1 < indirect_zones; i1++)
730075              {
730076                if ((inode->blkcnt > (7+indirect_zones+indirect_zones*i1))
730077                    && zone_table1[i1] != 0)
730078                  {
730079                    //
730080                    // There is a second level table to load.
730081                    //
730082                    status = zone_read (inode->sb, zone_table1[i1],
730083                                        zone_table2);
730084                    if (status == 0)
730085                      {
730086                        //
730087                        // Release zones.
730088                        //
730089                        for (i2 = 0;
730090                             i2 < indirect_zones &&
730091    (inode->blkcnt > (7+indirect_zones+indirect_zones*i1+i2));
730092                             i2++)
730093                          {
730094                            zone_free (inode->sb, zone_table2[i2]);
730095                          }
730096                        //
730097                        // Remove second level indirect table.
730098                        //
730099                        zone_free (inode->sb, zone_table1[i1]);
730100                      }
730101                  }
730102              }
730103            //
730104            // Remove first level indirect table.
730105            //
730106            zone_free (inode->sb, inode->indirect2);
730107          }
730108        //
730109        // Clear single indirect reference inside the inode.
730110        //
730111        inode->indirect2 = 0;
730112      }
730113    //
730114    // Update super block and inode data.
730115    //
730116    sb_save (inode->sb);
730117    inode->size = 0;
730118    inode->changed = 1;
730119    inode_save (inode);
730120    //
```

```
730121    // Successful return.
730122    //
730123    return (0);
730124  }
```

100.4.31 kernel/fs/inode_zone.c

Si veda la sezione 99.3.29.

```
740001  #include <kernel/fs.h>
740002  #include <errno.h>
740003  #include <kernel/k_libc.h>
740004  //----------------------------------------------------------------------
740005  zno_t
740006  inode_zone (inode_t *inode, zno_t fzone, int write)
740007  {
740008    unsigned int indirect_zones;
740009    unsigned int allocated_zone;
740010    zno_t        zone_table[INODE_MAX_INDIRECT_ZONES];
740011    char         buffer[SB_MAX_ZONE_SIZE];
740012    unsigned int i0;            // Single indirect index.
740013    unsigned int i1;            // Double indirect first index.
740014    unsigned int i2;            // Double indirect second index.
740015    int          status;
740016    zno_t        zone_second;   // Second level table zone.
740017    //
740018    // Calculate how many indirect zone numbers are stored inside
740019    // a zone: it depends on the zone size.
740020    //
740021    indirect_zones  = inode->sb->blksize / 2;
740022    //
740023    // Convert file-zone number into a zone number.
740024    //
740025    if (fzone < 7)
740026      {
740027        //
740028        // 0 <= fzone <= 6
740029        // The zone number is inside the direct zone references.
740030        // Verify to have such zone.
740031        //
740032        if (inode->direct[fzone] == 0)
740033          {
740034            //
740035            // There is not such zone, but we do not consider
740036            // it an error, because a file can be not contiguous.
740037            //
740038            if (!write)
740039              {
```

```
740040                  return ((zno_t) 0);
740041                }
740042            //
740043            // Must be allocated.
740044            //
740045            allocated_zone = zone_alloc (inode->sb);
740046            if (allocated_zone == 0)
740047              {
740048                //
740049                // Cannot allocate the zone. The variable 'errno' is
740050                // set by 'zone_alloc()'.
740051                //
740052                return ((zno_t) -1);
740053              }
740054            //
740055            // The zone is allocated: clear the zone and save.
740056            //
740057            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740058            status = zone_write (inode->sb, allocated_zone, buffer);
740059            if (status < 0)
740060              {
740061                //
740062                // Cannot overwrite the zone. The variable 'errno' is
740063                // set by 'zone_write()'.
740064                //
740065                return ((zno_t) -1);
740066              }
740067            //
740068            // The zone is allocated and cleared: save the inode.
740069            //
740070            inode->direct[fzone] = allocated_zone;
740071            inode->changed = 1;
740072            status = inode_save (inode);
740073            if (status != 0)
740074              {
740075                //
740076                // Cannot save the inode. The variable 'errno' is
740077                // set 'inode_save()'.
740078                //
740079                return ((zno_t) -1);
740080              }
740081          }
740082        //
740083        // The zone is there: return it.
740084        //
740085        return (inode->direct[fzone]);
740086      }
740087    if (fzone < 7 + indirect_zones)
```

```
740088      {
740089        //
740090        // 7 <= fzone <= (6 + indirect_zones)
740091        // The zone number is inside the single indirect zone
740092        // references: verify to have the indirect zone table.
740093        //
740094        if (inode->indirect1 == 0)
740095          {
740096            //
740097            // There is not such zone, but it is not an error.
740098            //
740099            if (!write)
740100              {
740101                return ((zno_t) 0);
740102              }
740103            //
740104            // The first level of indirection must be initialized.
740105            //
740106            allocated_zone = zone_alloc (inode->sb);
740107            if (allocated_zone == 0)
740108              {
740109                //
740110                // Cannot allocate the zone for the indirection table:
740111                // this is an error and the 'errno' value is produced
740112                // by 'zone_alloc()'.
740113                //
740114                return ((zno_t) -1);
740115              }
740116            //
740117            // The zone for the indirection table is allocated:
740118            // clear the zone and save.
740119            //
740120            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740121            status = zone_write (inode->sb, allocated_zone, buffer);
740122            if (status < 0)
740123              {
740124                //
740125                // Cannot overwrite the zone. The variable 'errno' is
740126                // set by 'zone_write()'.
740127                //
740128                return ((zno_t) -1);
740129              }
740130            //
740131            // The indirection table zone is allocated and cleared:
740132            // save the inode.
740133            //
740134            inode->indirect1 = allocated_zone;
740135            inode->changed = 1;
```

```
740136              status = inode_save (inode);
740137              if (status != 0)
740138                {
740139                  //
740140                  // Cannot save the inode. This is an error and the value
740141                  // for ‘errno’ is produced by ‘inode_save()’.
740142                  //
740143                  return ((zno_t) -1);
740144                }
740145            }
740146          //
740147          // An indirect table is present inside the file system:
740148          // load it.
740149          //
740150          status = zone_read (inode->sb, inode->indirect1, zone_table);
740151          if (status != 0)
740152            {
740153              //
740154              // Cannot load the indirect table. This is an error and the
740155              // value for ‘errno’ is assigned by function ‘zone_read()’.
740156              //
740157              return ((zno_t) -1);
740158            }
740159          //
740160          // The indirect table was read. Calculate the index inside
740161          // the table, for the requested zone.
740162          //
740163          i0 = (fzone - 7);
740164          //
740165          // Check if the zone is to be allocated.
740166          //
740167          if (zone_table[i0] == 0)
740168            {
740169              //
740170              // There is not such zone, but it is not an error.
740171              //
740172              if (!write)
740173                {
740174                  return ((zno_t) 0);
740175                }
740176              //
740177              // The zone must be allocated.
740178              //
740179              allocated_zone = zone_alloc (inode->sb);
740180              if (allocated_zone == 0)
740181                {
740182                  //
740183                  // There is no space for the zone allocation. The
```

```
740184                  // variable 'errno' is already updated by
740185                  // 'zone_alloc()'.
740186                  //
740187                  return ((zno_t) -1);
740188                }
740189              //
740190              // The zone is allocated: clear the zone and save.
740191              //
740192              memset (buffer, 0, SB_MAX_ZONE_SIZE);
740193              status = zone_write (inode->sb, allocated_zone, buffer);
740194              if (status < 0)
740195                {
740196                  //
740197                  // Cannot overwrite the zone. The variable 'errno' is
740198                  // set by 'zone_write()'.
740199                  //
740200                  return ((zno_t) -1);
740201                }
740202              //
740203              // The zone is allocated and cleared: update the indirect
740204              // zone table an save it. The inode is not modified,
740205              // because the indirect table is outside.
740206              //
740207              zone_table[i0] = allocated_zone;
740208              status = zone_write (inode->sb, inode->indirect1, zone_table);
740209              if (status != 0)
740210                {
740211                  //
740212                  // Cannot save the zone. The variable 'errno' is already
740213                  // set by 'zone_write()'.
740214                  //
740215                  return ((zno_t) -1);
740216                }
740217            }
740218          //
740219          // The zone is allocated.
740220          //
740221          return (zone_table[i0]);
740222        }
740223      else
740224        {
740225          //
740226          // (7 + indirect_zones) <= fzone
740227          // The zone number is inside the double indirect zone
740228          // references.
740229          // Verify to have the first level of second indirection.
740230          //
740231          if (inode->indirect2 == 0)
```

```
740232          {
740233            //
740234            // There is not such zone, but it is not an error.
740235            //
740236            if (!write)
740237              {
740238                return ((zno_t) 0);
740239              }
740240            //
740241            // The first level of second indirection must be
740242            // initialized.
740243            //
740244            allocated_zone = zone_alloc (inode->sb);
740245            if (allocated_zone == 0)
740246              {
740247                //
740248                // Cannot allocate the zone. The variable ‘errno’ is
740249                // set by ‘zone_alloc()’.
740250                //
740251                return ((zno_t) -1);
740252              }
740253            //
740254            // The zone for the indirection table is allocated:
740255            // clear the zone and save.
740256            //
740257            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740258            status = zone_write (inode->sb, allocated_zone, buffer);
740259            if (status < 0)
740260              {
740261                //
740262                // Cannot overwrite the zone. The variable ‘errno’ is
740263                // set by ‘zone_write()’.
740264                //
740265                return ((zno_t) -1);
740266              }
740267            //
740268            // The zone for the indirection table is allocated and
740269            // cleared: save the inode.
740270            //
740271            inode->indirect2 = allocated_zone;
740272            inode->changed = 1;
740273            status = inode_save (inode);
740274            if (status != 0)
740275              {
740276                //
740277                // Cannot save the inode. The variable ‘errno’ is
740278                // set by ‘inode_save()’.
740279                //
```

```
740280                  return ((zno_t) -1);
740281                }
740282            }
740283          //
740284          // The first level of second indirection is present:
740285          // Read the second indirect table.
740286          //
740287          status = zone_read (inode->sb, inode->indirect2, zone_table);
740288          if (status != 0)
740289            {
740290              //
740291              // Cannot read the second indirect table. The variable
740292              // `errno' is set by `zone_read()'.
740293              //
740294              return ((zno_t) -1);
740295            }
740296          //
740297          // The first double indirect table was read: calculate
740298          // indexes inside first and second level of table.
740299          //
740300          fzone -= 7;
740301          fzone -= indirect_zones;
740302          i1     = fzone / indirect_zones;
740303          i2     = fzone % indirect_zones;
740304          //
740305          // Verify to have a second level.
740306          //
740307          if (zone_table[i1] == 0)
740308            {
740309              //
740310              // There is not such zone, but it is not an error.
740311              //
740312              if (!write)
740313                {
740314                  return ((zno_t) 0);
740315                }
740316              //
740317              // The second level must be initialized.
740318              //
740319              allocated_zone = zone_alloc (inode->sb);
740320              if (allocated_zone == 0)
740321                {
740322                  //
740323                  // Cannot allocate the zone. The variable `errno' is set
740324                  // by `zone_alloc()'.
740325                  //
740326                  return ((zno_t) -1);
740327                }
```

```
740328                //
740329                // The zone for the indirection table is allocated:
740330                // clear the zone and save.
740331                //
740332                memset (buffer, 0, SB_MAX_ZONE_SIZE);
740333                status = zone_write (inode->sb, allocated_zone, buffer);
740334                if (status < 0)
740335                  {
740336                    //
740337                    // Cannot overwrite the zone. The variable 'errno' is
740338                    // set by 'zone_write()'.
740339                    //
740340                    return ((zno_t) -1);
740341                  }
740342                //
740343                // Update the first level index and save it.
740344                //
740345                zone_table[i1] = allocated_zone;
740346                status = zone_write (inode->sb, inode->indirect2, zone_table);
740347                if (status != 0)
740348                  {
740349                    //
740350                    // Cannot write the zone. The variable 'errno' is set
740351                    // by 'zone_write()'.
740352                    //
740353                    return ((zno_t) -1);
740354                  }
740355              }
740356            //
740357            // The second level can be read, overwriting the array
740358            // 'zone_table[]'. The zone number for the second level
740359            // indirection table is saved inside 'zone_second', before
740360            // overwriting the array.
740361            //
740362            zone_second = zone_table[i1];
740363            status = zone_read (inode->sb, zone_second, zone_table);
740364            if (status != 0)
740365              {
740366                //
740367                // Cannot read the second level indirect table. The variable
740368                // 'errno' is set by 'zone_read()'.
740369                //
740370                return ((zno_t) -1);
740371              }
740372            //
740373            // The second level was read and 'zone_table[]' is now
740374            // such second one: check if the zone is to be allocated.
740375            //
```

```
740376        if (zone_table[i2] == 0)
740377          {
740378            //
740379            // There is not such zone, but it is not an error.
740380            //
740381            if (!write)
740382              {
740383                return ((zno_t) 0);
740384              }
740385            //
740386            // Must be allocated.
740387            //
740388            allocated_zone = zone_alloc (inode->sb);
740389            if (allocated_zone == 0)
740390              {
740391                //
740392                // Cannot allocate the zone. The variable 'errno' is set
740393                // by 'zone_alloc()'.
740394                //
740395                return ((zno_t) -1);
740396              }
740397            //
740398            // The zone is allocated: clear the zone and save.
740399            //
740400            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740401            status = zone_write (inode->sb, allocated_zone, buffer);
740402            if (status < 0)
740403              {
740404                //
740405                // Cannot overwrite the zone. The variable 'errno' is
740406                // set by 'zone_write()'.
740407                //
740408                return ((zno_t) -1);
740409              }
740410            //
740411            // The zone was allocated and cleared: update the indirect
740412            // zone table an save it. The inode is not modified, because
740413            // the indirect table is outside.
740414            //
740415            zone_table[i2] = allocated_zone;
740416            status = zone_write (inode->sb, zone_second, zone_table);
740417            if (status != 0)
740418              {
740419                //
740420                // Cannot write the zone. The variable 'errno' is set
740421                // by 'zone_write()'.
740422                //
740423                return ((zno_t) -1);
```

```
740424                }
740425              }
740426            //
740427            // The zone is there: return the zone number.
740428            //
740429            return (zone_table[i2]);
740430          }
740431  }
```

100.4.32 kernel/fs/path_chdir.c

«

Si veda la sezione 99.3.30.

```
750001  #include <kernel/fs.h>
750002  #include <errno.h>
750003  #include <kernel/proc.h>
750004  //----------------------------------------------------------------------
750005  int
750006  path_chdir (pid_t pid, const char *path)
750007  {
750008      proc_t   *ps;
750009      inode_t  *inode_directory;
750010      int       status;
750011      char      path_directory[PATH_MAX];
750012      //
750013      // Get process.
750014      //
750015      ps = proc_reference (pid);
750016      //
750017      // The full directory path is needed.
750018      //
750019      status = path_full (path, ps->path_cwd, path_directory);
750020      if (status < 0)
750021        {
750022          return (-1);
750023        }
750024      //
750025      // Try to load the new directory inode.
750026      //
750027      inode_directory = path_inode (pid, path_directory);
750028      if (inode_directory == NULL)
750029        {
750030          //
750031          // Cannot access the directory: it does not exists or
750032          // permissions are not sufficient. Variable 'errno' is set by
750033          // function 'inode_directory()'.
750034          //
750035          errset (errno);
```

```
750036             return (-1);
750037           }
750038       //
750039       // Inode loaded: release the old directory and set the new one.
750040       //
750041       inode_put (ps->inode_cwd);
750042       //
750043       ps->inode_cwd = inode_directory;
750044       strncpy (ps->path_cwd, path_directory, PATH_MAX);
750045       //
750046       // Return.
750047       //
750048       return (0);
750049   }
```

100.4.33 kernel/fs/path_chmod.c

«

Si veda la sezione 99.3.31.

```
760001   #include <kernel/fs.h>
760002   #include <errno.h>
760003   #include <kernel/proc.h>
760004   //----------------------------------------------------------------------
760005   int
760006   path_chmod (pid_t pid, const char *path, mode_t mode)
760007   {
760008       proc_t   *ps;
760009       inode_t  *inode;
760010       //
760011       // Get process.
760012       //
760013       ps = proc_reference (pid);
760014       //
760015       // Try to load the file inode.
760016       //
760017       inode = path_inode (pid, path);
760018       if (inode == NULL)
760019         {
760020           //
760021           // Cannot access the file: it does not exists or permissions are
760022           // not sufficient. Variable `errno' is set by function
760023           // `inode_directory()'.
760024           //
760025           return (-1);
760026         }
760027       //
760028       // Verify to be root or to be the owner.
760029       //
```

```
760030      if (ps->euid != 0 && ps->euid != inode->uid)
760031        {
760032          errset (EACCES);         // Permission denied.
760033          return (-1);
760034        }
760035      //
760036      // Update the mode: the file type is kept and the
760037      // rest is taken form the parameter 'mode'.
760038      //
760039      inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
760040      //
760041      // Save and release the inode.
760042      //
760043      inode->changed = 1;
760044      inode_save (inode);
760045      inode_put (inode);
760046      //
760047      // Return.
760048      //
760049      return (0);
760050  }
```

100.4.34 kernel/fs/path_chown.c

«

Si veda la sezione 99.3.32.

```
770001  #include <kernel/fs.h>
770002  #include <errno.h>
770003  #include <kernel/proc.h>
770004  //----------------------------------------------------------------------
770005  int
770006  path_chown (pid_t pid, const char *path, uid_t uid, gid_t gid)
770007  {
770008      proc_t   *ps;
770009      inode_t  *inode;
770010      //
770011      // Get process.
770012      //
770013      ps = proc_reference (pid);
770014      //
770015      // Must be root, as the ability to change group is not considered.
770016      //
770017      if (ps->euid != 0)
770018        {
770019          errset (EPERM);                 // Operation not permitted.
770020          return (-1);
770021        }
770022      //
```

```
770023     // Try to load the file inode.
770024     //
770025     inode = path_inode (pid, path);
770026     if (inode == NULL)
770027       {
770028         //
770029         // Cannot access the file: it does not exists or permissions are
770030         // not sufficient. Variable 'errno' is set by function
770031         // 'inode_directory()'.
770032         //
770033         return (-1);
770034       }
770035     //
770036     // Update the owner and group.
770037     //
770038     if (uid != -1)
770039       {
770040         inode->uid     = uid;
770041         inode->changed = 1;
770042       }
770043     if (gid != -1)
770044       {
770045         inode->gid     = gid;
770046         inode->changed = 1;
770047       }
770048     //
770049     // Save and release the inode.
770050     //
770051     inode_save (inode);
770052     inode_put (inode);
770053     //
770054     // Return.
770055     //
770056     return (0);
770057 }
```

100.4.35 kernel/fs/path_device.c

Si veda la sezione 99.3.33.

```
780001 #include <kernel/fs.h>
780002 #include <errno.h>
780003 #include <kernel/proc.h>
780004 //----------------------------------------------------------------------
780005 dev_t
780006 path_device (pid_t pid, const char *path)
780007 {
780008     proc_t   *ps;
```

```
780009      inode_t   *inode;
780010      dev_t      device;
780011      //
780012      // Get process.
780013      //
780014      ps = proc_reference (pid);
780015      //
780016      inode = path_inode (pid, path);
780017      if (inode == NULL)
780018        {
780019          errset (errno);
780020          return ((dev_t) -1);
780021        }
780022      //
780023      if (!(S_ISBLK (inode->mode) || S_ISCHR (inode->mode)))
780024        {
780025          errset (ENODEV);                    // No such device.
780026          inode_put (inode);
780027          return ((dev_t) -1);
780028        }
780029      //
780030      device = inode->direct[0];
780031      inode_put (inode);
780032      return (device);
780033  }
```

100.4.36 kernel/fs/path_fix.c

«

Si veda la sezione 99.3.34.

```
790001  #include <kernel/fs.h>
790002  #include <errno.h>
790003  #include <kernel/proc.h>
790004  //----------------------------------------------------------------------
790005  int
790006  path_fix (char *path)
790007  {
790008      char    new_path[PATH_MAX];
790009      char   *token[PATH_MAX/4];
790010      int     t;                            // Token index.
790011      int     token_size;                   // Token array effective size.
790012      int     comp;                         // String compare return value.
790013      size_t  path_size;                    // Path string size.
790014      //
790015      // Initialize token search.
790016      //
790017      token[0] = strtok (path, "/");
790018      //
```

```
790019      // Scan tokens.
790020      //
790021      for (t = 0;
790022           t < PATH_MAX/4 && token[t] != NULL;
790023           t++, token[t] = strtok (NULL, "/"))
790024        {
790025          //
790026          // If current token is `.', just ignore it.
790027          //
790028          comp = strcmp (token[t], ".");
790029          if (comp == 0)
790030            {
790031              t--;
790032            }
790033          //
790034          // If current token is `..', remove previous token,
790035          // if there is one.
790036          //
790037          comp = strcmp (token[t], "..");
790038          if (comp == 0)
790039            {
790040              if (t > 0)
790041                {
790042                  t -= 2;
790043                }
790044              else
790045                {
790046                  t = -1;
790047                }
790048            }
790049          //
790050          // `t' will be incremented and another token will be
790051          // found.
790052          //
790053        }
790054      //
790055      // Save the token array effective size.
790056      //
790057      token_size = t;
790058      //
790059      // Initialize the new path string.
790060      //
790061      new_path[0] = '\0';
790062      //
790063      // Build the new path string.
790064      //
790065      if (token_size > 0)
790066        {
```

```
790067          for (t = 0; t < token_size; t++)
790068            {
790069              path_size = strlen (new_path);
790070              strncat (new_path, "/", 2);
790071              strncat (new_path, token[t], PATH_MAX - path_size - 1);
790072            }
790073        }
790074      else
790075        {
790076          strncat (new_path, "/", 2);
790077        }
790078      //
790079      // Copy the new path into the original string.
790080      //
790081      strncpy (path, new_path, PATH_MAX);
790082      //
790083      // Return.
790084      //
790085      return (0);
790086  }
```

100.4.37 kernel/fs/path_full.c

«

Si veda la sezione 99.3.35.

```
800001  #include <kernel/fs.h>
800002  #include <errno.h>
800003  #include <kernel/proc.h>
800004  //----------------------------------------------------------------------
800005  int
800006  path_full (const char *path, const char *path_cwd, char *full_path)
800007  {
800008      unsigned int path_size;
800009      //
800010      // Check some arguments.
800011      //
800012      if (path == NULL || strlen (path) == 0 || full_path == NULL)
800013        {
800014          errset (EINVAL);                  // Invalid argument.
800015          return (-1);
800016        }
800017      //
800018      // The main path and the receiving one are right.
800019      // Now arrange to get a full path name.
800020      //
800021      if (path[0] == '/')
800022        {
800023          strncpy (full_path, path, PATH_MAX);
```

```
800024            full_path[PATH_MAX-1] = 0;
800025          }
800026        else
800027          {
800028            if (path_cwd == NULL || strlen (path_cwd) == 0)
800029              {
800030                errset (EINVAL);            // Invalid argument.
800031                return (-1);
800032              }
800033            strncpy (full_path, path_cwd, PATH_MAX);
800034            path_size = strlen (full_path);
800035            strncat (full_path, "/", (PATH_MAX - path_size));
800036            path_size = strlen (full_path);
800037            strncat (full_path, path, (PATH_MAX - path_size));
800038          }
800039        //
800040        // Fix path name so that it has no `..', `.', and no
800041        // multiple `/'.
800042        //
800043        path_fix (full_path);
800044        //
800045        // Return.
800046        //
800047        return (0);
800048    }
```

100.4.38 kernel/fs/path_inode.c

«

Si veda la sezione 99.3.36.

```
810001    #include <kernel/fs.h>
810002    #include <errno.h>
810003    #include <kernel/proc.h>
810004    #include <kernel/k_libc.h>
810005    //----------------------------------------------------------------------
810006    #define DIRECTORY_BUFFER_SIZE (SB_MAX_ZONE_SIZE/16)
810007    //----------------------------------------------------------------------
810008    inode_t *
810009    path_inode (pid_t pid, const char *path)
810010    {
810011        proc_t      *ps;
810012        inode_t     *inode;
810013        dev_t        device;
810014        char         full_path[PATH_MAX];
810015        char        *name;
810016        char        *next;
810017        directory_t dir[DIRECTORY_BUFFER_SIZE];
810018        char         dir_name[NAME_MAX+1];
```

```
810019      off_t       offset_dir;
810020      ssize_t     size_read;
810021      size_t      dir_size_read;
810022      ssize_t     size_to_read;
810023      int         comp;
810024      int         d;                // Directory index;
810025      int         status;           // inode_check() return status.
810026      //
810027      // Get process.
810028      //
810029      ps = proc_reference (pid);
810030      //
810031      // Arrange to get a packed full path name.
810032      //
810033      path_full (path, ps->path_cwd, full_path);
810034      //
810035      // Get the root file system inode.
810036      //
810037      inode = inode_get ((dev_t) 0, 1);
810038      if (inode == NULL)
810039        {
810040          errset (errno);
810041          return (NULL);
810042        }
810043      //
810044      // Save the device number.
810045      //
810046      device = inode->sb->device;
810047      //
810048      // Variable 'inode' already points to the root file system inode:
810049      // It must be a directory!
810050      //
810051      status = inode_check (inode, S_IFDIR, 1, ps->euid);
810052      if (status != 0)
810053        {
810054          //
810055          // Variable 'errno' should be set by inode_check().
810056          //
810057          errset (errno);
810058          inode_put (inode);
810059          return (NULL);
810060        }
810061      //
810062      // Initialize string scan: find the first path token, after the
810063      // first '/'.
810064      //
810065      name = strtok (full_path, "/");
810066      //
```

```
810067      // If the original full path is just '/' the variable 'name'
810068      // appears as a null pointer, and the variable 'inode' is already
810069      // what we are looking for.
810070      //
810071      if (name == NULL)
810072        {
810073          return (inode);
810074        }
810075      //
810076      // There is at least a name after '/' inside the original full
810077      // path. A scan is going to start: the original value for variable
810078      // 'inode' is a pointer to the root directory inode.
810079      //
810080      for (;;)
810081        {
810082          //
810083          // Find next token.
810084          //
810085          next = strtok (NULL, "/");
810086          //
810087          // Read the directory from the current inode.
810088          //
810089          for (offset_dir=0; ; offset_dir += size_read)
810090            {
810091              size_to_read = DIRECTORY_BUFFER_SIZE;
810092              //
810093              if ((offset_dir + size_to_read) > inode->size)
810094                {
810095                  size_to_read = inode->size - offset_dir;
810096                }
810097              //
810098              size_read = inode_file_read (inode, offset_dir, dir,
810099                                           size_to_read, NULL);
810100              //
810101              // The size read must be a multiple of 16.
810102              //
810103              size_read = ((size_read / 16) * 16);
810104              //
810105              // Check anyway if it is zero.
810106              //
810107              if (size_read == 0)
810108                {
810109                  //
810110                  // The directory is ended: release the inode and return.
810111                  //
810112                  inode_put (inode);
810113                  errset (ENOENT);           // No such file or directory.
810114                  return (NULL);
```

```
810115                  }
810116                //
810117                // Calculate how many directory items we have read.
810118                //
810119                dir_size_read = size_read / 16;
810120                //
810121                // Scan the directory to find the current name.
810122                //
810123                for (d = 0; d < dir_size_read; d++)
810124                  {
810125                    //
810126                    // Ensure to have a null terminated string for
810127                    // the name found.
810128                    //
810129                    memcpy (dir_name, dir[d].name, (size_t) NAME_MAX);
810130                    dir_name[NAME_MAX] = 0;
810131                    //
810132                    comp = strcmp (name, dir_name);
810133                    if (comp == 0 && dir[d].ino != 0)
810134                      {
810135                        //
810136                        // Found the name and verified that it has a link to
810137                        // a inode. Now release the directory inode.
810138                        //
810139                        inode_put (inode);
810140                        //
810141                        // Get next inode and break the loop.
810142                        //
810143                        inode = inode_get (device, dir[d].ino);
810144                        break;
810145                      }
810146                  }
810147                //
810148                // If index 'd' is in a valid range, the name was found.
810149                //
810150                if (d < dir_size_read)
810151                  {
810152                    //
810153                    // The name was found.
810154                    //
810155                    break;
810156                  }
810157              }
810158            //
810159            // If the function is still working, a file or a directory
810160            // was found: see if there is another name after this one
810161            // to look for. If there isn't, just break the loop.
810162            //
```

```
810163          if (next == NULL)
810164            {
810165              //
810166              // As no other tokens are to be found, break the loop.
810167              //
810168              break;
810169            }
810170          //
810171          // As there is another name after the current one,
810172          // the current file must be a directory.
810173          //
810174          status = inode_check (inode, S_IFDIR, 1, ps->euid);
810175          if (status != 0)
810176            {
810177              //
810178              // Variable 'errno' is set by 'inode_check()'.
810179              //
810180              errset (errno);
810181              inode_put (inode);
810182              return (NULL);
810183           }
810184          //
810185          // The inode is a directory and the user has the necessary
810186          // permissions: check if it is a mount point and go to the
810187          // new device root directory if necessary.
810188          //
810189          if (inode->sb_attached != NULL)
810190            {
810191              //
810192              // Must find the root directory for the new device, and
810193              // then go to that inode.
810194              //
810195              device = inode->sb_attached->device;
810196              inode_put (inode);
810197              inode = inode_get (device, 1);
810198              status = inode_check (inode, S_IFDIR, 1, ps->euid);
810199              if (status != 0)
810200                {
810201                  inode_put (inode);
810202                  return (NULL);
810203                }
810204            }
810205          //
810206          // As a directory was found, and another token follows it,
810207          // must continue the token scan.
810208          //
810209          name = next;
810210        }
```

```
810211     //
810212     // Current inode found is the file represented by the requested
810213     // path.
810214     //
810215     return (inode);
810216 }
```

100.4.39 kernel/fs/path_inode_link.c

«

Si veda la sezione 99.3.37.

```
820001 #include <kernel/fs.h>
820002 #include <errno.h>
820003 #include <kernel/proc.h>
820004 #include <libgen.h>
820005 //----------------------------------------------------------------------
820006 inode_t *
820007 path_inode_link (pid_t pid, const char *path, inode_t *inode,
820008                  mode_t mode)
820009 {
820010     proc_t      *ps;
820011     char         buffer[SB_MAX_ZONE_SIZE];
820012     off_t        start;
820013     int          d;        // Directory index.
820014     ssize_t      size_read;
820015     ssize_t      size_written;
820016     directory_t *dir = (directory_t *) buffer;
820017     char         path_copy1[PATH_MAX];
820018     char         path_copy2[PATH_MAX];
820019     char        *path_directory;
820020     char        *path_name;
820021     inode_t     *inode_directory;
820022     inode_t     *inode_new;
820023     dev_t        device;
820024     int          status;
820025     //
820026     // Check arguments.
820027     //
820028     if (path == NULL || strlen (path) == 0)
820029       {
820030         errset (EINVAL);         // Invalid argument:
820031         return (NULL);           // the path is mandatory.
820032       }
820033     //
820034     if (inode == NULL && mode == 0)
820035       {
820036         errset (EINVAL);         // Invalid argument: if the inode is to
820037         return (NULL);           // be created, the mode is mandatory.
```

```
820038            }
820039        //
820040        if (inode != NULL)
820041          {
820042            if (mode != 0)
820043              {
820044                errset (EINVAL);     // Invalid argument: if the inode is
820045                return (NULL);       // already present, the creation mode
820046              }                      // must not be given.
820047            if (S_ISDIR (inode->mode))
820048              {
820049                errset (EPERM);      // Operation not permitted.
820050                return (NULL);       // Refuse to link directory.
820051              }
820052            if (inode->links >= LINK_MAX)
820053              {
820054                errset (EMLINK);             // Too many links.
820055                return (NULL);
820056              }
820057          }
820058        //
820059        // Get process.
820060        //
820061        ps = proc_reference (pid);
820062        //
820063        // If the destination path already exists, the link cannot be made.
820064        // It does not matter if the inode is known or not.
820065        //
820066        inode_new = path_inode ((uid_t) 0, path);
820067        if (inode_new != NULL)
820068          {
820069            //
820070            // A file already exists with the same name.
820071            //
820072            inode_put (inode_new);
820073            errset (EEXIST);                 // File exists.
820074            return (NULL);
820075          }
820076        //
820077        // At this point, 'inode_new' is 'NULL'.
820078        // Copy the source path inside the directory path and name arrays.
820079        //
820080        strncpy (path_copy1, path, PATH_MAX);
820081        strncpy (path_copy2, path, PATH_MAX);
820082        //
820083        // Reduce to directory name and find the last name.
820084        //
820085        path_directory = dirname  (path_copy1);
```

```
820086      path_name       = basename (path_copy2);
820087      if (strlen (path_directory) == 0 || strlen (path_name) == 0)
820088        {
820089          errset (EACCES);        // Permission denied: maybe the
820090                                  // original path is the root directory
820091                                  // and cannot find a previous directory.
820092          return (NULL);
820093        }
820094      //
820095      // Get the directory inode.
820096      //
820097      inode_directory = path_inode (pid, path_directory);
820098      if (inode_directory == NULL)
820099        {
820100          errset (errno);
820101          return (NULL);
820102        }
820103      //
820104      // Check if something is mounted on it.
820105      //
820106      if (inode_directory->sb_attached != NULL)
820107        {
820108          //
820109          // Must select the right directory.
820110          //
820111          device = inode_directory->sb_attached->device;
820112          inode_put (inode_directory);
820113          inode_directory = inode_get (device, 1);
820114          if (inode_directory == NULL)
820115            {
820116              return (NULL);
820117            }
820118        }
820119      //
820120      // If the inode to link is known, check if the selected directory
820121      // has the same super block than the inode to link.
820122      //
820123      if (inode != NULL && inode_directory->sb != inode->sb)
820124        {
820125          inode_put (inode_directory);
820126          errset (ENOENT);        // No such file or directory.
820127          return (NULL);
820128        }
820129      //
820130      // Check if write is allowed for the file system.
820131      //
820132      if (inode_directory->sb->options & MOUNT_RO)
820133        {
```

```
820134             inode_put (inode_directory);
820135             errset (EROFS);          // Read-only file system.
820136             return (NULL);
820137           }
820138         //
820139         // Verify access permissions for the directory. The number "3" means
820140         // that the user must have access permission and write permission:
820141         // "-wx" == 2+1 == 3.
820142         //
820143         status = inode_check (inode_directory, S_IFDIR, 3, ps->euid);
820144         if (status != 0)
820145           {
820146             inode_put (inode_directory);
820147             return (NULL);
820148           }
820149         //
820150         // If the inode to link was not specified, it must be created.
820151         // From now on, the inode is referenced with the variable
820152         // 'inode_new'.
820153         //
820154         inode_new = inode;
820155         //
820156         if (inode_new == NULL)
820157           {
820158             inode_new = inode_alloc (inode_directory->sb->device, mode,
820159                                      ps->euid);
820160             if (inode_new == NULL)
820161               {
820162                 //
820163                 // The inode allocation failed, so, also the directory
820164                 // must be released, before return.
820165                 //
820166                 inode_put (inode_directory);
820167                 return (NULL);
820168               }
820169           }
820170         //
820171         // Read the directory content and try to add the new item.
820172         //
820173         for (start  = 0;
820174              start <  inode_directory->size;
820175              start += inode_directory->sb->blksize)
820176           {
820177             size_read = inode_file_read (inode_directory, start, buffer,
820178                                          inode_directory->sb->blksize,
820179                                          NULL);
820180             if (size_read < sizeof (directory_t))
820181               {
```

```
820182              break;
820183            }
820184          //
820185          // Scan the directory portion just read, for an unused item.
820186          //
820187          dir = (directory_t *) buffer;
820188          for (d = 0; d < size_read; d += (sizeof (directory_t)), dir++)
820189            {
820190              if (dir->ino == 0)
820191                {
820192                  //
820193                  // Found an empty directory item: link the inode.
820194                  //
820195                  dir->ino = inode_new->ino;
820196                  strncpy (dir->name, path_name, NAME_MAX);
820197                  inode_new->links++;
820198                  inode_new->changed = 1;
820199                  //
820200                  // Update the directory inside the file system.
820201                  //
820202                  size_written = inode_file_write (inode_directory, start,
820203                                                   buffer, size_read);
820204                  if (size_written != size_read)
820205                    {
820206                      //
820207                      // Write problem: release the directory and return.
820208                      //
820209                      inode_put (inode_directory);
820210                      errset (EUNKNOWN);
820211                      return (NULL);
820212                    }
820213                  //
820214                  // Save the new inode, release the directory and return
820215                  // the linked inode.
820216                  //
820217                  inode_save (inode_new);
820218                  inode_put (inode_directory);
820219                  return (inode_new);
820220                }
820221            }
820222        }
820223      //
820224      // The directory don't have a free item and one must be appended.
820225      //
820226      dir   = (directory_t *) buffer;
820227      start = inode_directory->size;
820228      //
820229      // Prepare the buffer with the link.
```

```
820230     //
820231     dir->ino = inode_new->ino;
820232     strncpy (dir->name, path_name, NAME_MAX);
820233     inode_new->links++;
820234     inode_new->changed = 1;
820235     //
820236     // Append the buffer to the directory.
820237     //
820238     size_written = inode_file_write (inode_directory, start, buffer,
820239                                      (sizeof (directory_t)));
820240     if (size_written != (sizeof (directory_t)))
820241       {
820242         //
820243         // Problem updating the directory: release it and return.
820244         //
820245         inode_put (inode_directory);
820246         errset (EUNKNOWN);
820247         return (NULL);
820248       }
820249     //
820250     // Close access to the directory inode and save the other inode,
820251     // with updated link count.
820252     //
820253     inode_put (inode_directory);
820254     inode_save (inode_new);
820255     //
820256     // Return successfully.
820257     //
820258     return (inode_new);
820259 }
```

100.4.40 kernel/fs/path_link.c

Si veda la sezione 99.3.38.

```
830001 #include <kernel/fs.h>
830002 #include <errno.h>
830003 #include <kernel/proc.h>
830004 //----------------------------------------------------------------------
830005 int
830006 path_link (pid_t pid, const char *path_old, const char *path_new)
830007 {
830008     proc_t              *ps;
830009     inode_t             *inode_old;
830010     inode_t             *inode_new;
830011     char                 path_new_full[PATH_MAX];
830012     //
830013     // Get process.
```

```
830014     //
830015     ps = proc_reference (pid);
830016     //
830017     // Try to get the old path inode.
830018     //
830019     inode_old = path_inode (pid, path_old);
830020     if (inode_old == NULL)
830021       {
830022         //
830023         // Cannot get the inode: 'errno' is already set by
830024         // 'path_inode()'.
830025         //
830026         errset (errno);
830027         return (-1);
830028       }
830029     //
830030     // The inode is available and checks are done: arrange to get a
830031     // packed full path name and then the destination directory path.
830032     //
830033     path_full (path_new, ps->path_cwd, path_new_full);
830034     //
830035     //
830036     //
830037     inode_new = path_inode_link (pid, path_new_full, inode_old,
830038                                  (mode_t) 0);
830039     if (inode_new == NULL)
830040       {
830041         inode_put (inode_old);
830042         return (-1);
830043       }
830044     if (inode_new != inode_old)
830045       {
830046         inode_put (inode_new);
830047         inode_put (inode_old);
830048         errset (EUNKNOWN);              // Unknown error.
830049         return (-1);
830050       }
830051     //
830052     // Inode data is already updated by 'path_inode_link()': just put
830053     // it and return. Please note that only one is put, because it is
830054     // just the same of the other.
830055     //
830056     inode_put  (inode_new);
830057     return (0);
830058 }
```

100.4.41 kernel/fs/path_mkdir.c

Si veda la sezione 99.3.39.

```
840001 #include <kernel/fs.h>
840002 #include <errno.h>
840003 #include <kernel/proc.h>
840004 #include <libgen.h>
840005 #include <kernel/k_libc.h>
840006 //----------------------------------------------------------------------
840007 int
840008 path_mkdir (pid_t pid, const char *path, mode_t mode)
840009 {
840010     proc_t   *ps;
840011     inode_t  *inode_directory;
840012     inode_t  *inode_parent;
840013     int       status;
840014     char      path_directory[PATH_MAX];
840015     char      path_copy[PATH_MAX];
840016     char     *path_parent;
840017     ssize_t   size_written;
840018     //
840019     struct {
840020         ino_t inode_1;
840021         char  name_1[NAME_MAX];
840022         ino_t inode_2;
840023         char  name_2[NAME_MAX];
840024     } directory;
840025     //
840026     // Get process.
840027     //
840028     ps = proc_reference (pid);
840029     //
840030     // Correct the mode with the umask.
840031     //
840032     mode &= ~ps->umask;
840033     //
840034     // Inside 'mode', the file type is fixed. No check is made.
840035     //
840036     mode &= 00777;
840037     mode |= S_IFDIR;
840038     //
840039     // The full path and the directory path is needed.
840040     //
840041     status = path_full (path, ps->path_cwd, path_directory);
840042     if (status < 0)
840043       {
840044         return (-1);
840045       }
```

```
840046      strncpy (path_copy, path_directory, PATH_MAX);
840047      path_copy[PATH_MAX-1] = 0;
840048      path_parent = dirname (path_copy);
840049      //
840050      // Check if something already exists with the same name. The scan
840051      // is done with kernel privileges.
840052      //
840053      inode_directory = path_inode ((uid_t) 0, path_directory);
840054      if (inode_directory != NULL)
840055        {
840056          //
840057          // The file already exists. Put inode and return an error.
840058          //
840059          inode_put (inode_directory);
840060          errset (EEXIST);                // File exists.
840061          return (-1);
840062        }
840063      //
840064      // Try to locate the directory that should contain this one.
840065      //
840066      inode_parent = path_inode (pid, path_parent);
840067      if (inode_parent == NULL)
840068        {
840069          //
840070          // Cannot locate the directory: return an error. The variable
840071          // 'errno' should already be set by 'path_inode()'.
840072          //
840073          errset (errno);
840074          return (-1);
840075        }
840076      //
840077      // Try to create the node: should fail if the user does not have
840078      // enough permissions.
840079      //
840080      inode_directory = path_inode_link (pid, path_directory, NULL,
840081                                         mode);
840082      if (inode_directory == NULL)
840083        {
840084          //
840085          // Sorry: cannot create the inode! The variable 'errno' should
840086          // already be set by 'path_inode_link()'.
840087          //
840088          errset (errno);
840089          return (-1);
840090        }
840091      //
840092      // Fill records for '.' and '..'.
840093      //
```

```
840094      directory.inode_1 = inode_directory->ino;
840095      strncpy (directory.name_1, ".",  (size_t) 3);
840096      directory.inode_2 = inode_parent->ino;
840097      strncpy (directory.name_2, "..", (size_t) 3);
840098      //
840099      // Write data.
840100      //
840101      size_written = inode_file_write (inode_directory, (off_t) 0,
840102                                       &directory, (sizeof directory));
840103      if (size_written != (sizeof directory))
840104        {
840105          return (-1);
840106        }
840107      //
840108      // Fix directory inode links.
840109      //
840110      inode_directory->links   = 2;
840111      inode_directory->time    = k_time (NULL);
840112      inode_directory->changed = 1;
840113      //
840114      // Fix parent directory inode links.
840115      //
840116      inode_parent->links++;
840117      inode_parent->time       = k_time (NULL);
840118      inode_parent->changed    = 1;
840119      //
840120      // Save and put the inodes.
840121      //
840122      inode_save (inode_parent);
840123      inode_save (inode_directory);
840124      inode_put  (inode_parent);
840125      inode_put  (inode_directory);
840126      //
840127      // Return.
840128      //
840129      return (0);
840130  }
```

100.4.42 kernel/fs/path_mknod.c

Si veda la sezione 99.3.40.

```
850001  #include <kernel/fs.h>
850002  #include <errno.h>
850003  #include <kernel/proc.h>
850004  //----------------------------------------------------------------------
850005  int
850006  path_mknod (pid_t pid, const char *path, mode_t mode, dev_t device)
```

```
850007 {
850008     proc_t    *ps;
850009     inode_t   *inode;
850010     char       full_path[PATH_MAX];
850011     //
850012     // Get process.
850013     //
850014     ps = proc_reference (pid);
850015     //
850016     // Correct the mode with the umask.
850017     //
850018     mode &= ~ps->umask;
850019     //
850020     // Currently must be root for any kind of node to be created.
850021     //
850022     if (ps->uid != 0)
850023       {
850024         errset (EPERM);                 // Operation not permitted.
850025         return (-1);
850026       }
850027     //
850028     // Check the type of node requested.
850029     //
850030     if (!(S_ISBLK (mode) ||
850031           S_ISCHR (mode) ||
850032           S_ISREG (mode) ||
850033           S_ISDIR (mode)))
850034       {
850035         errset (EINVAL);                // Invalid argument.
850036         return (-1);
850037       }
850038     //
850039     // Check if something already exists with the same name.
850040     //
850041     inode = path_inode (pid, path);
850042     if (inode != NULL)
850043       {
850044         //
850045         // The file already exists. Put inode and return an error.
850046         //
850047         inode_put (inode);
850048         errset (EEXIST);                // File exists.
850049         return (-1);
850050       }
850051     //
850052     // Try to creat the node.
850053     //
850054     path_full (path, ps->path_cwd, full_path);
```

```
850055      inode = path_inode_link (pid, full_path, NULL, mode);
850056      if (inode == NULL)
850057        {
850058          //
850059          // Sorry: cannot create the inode!
850060          //
850061          return (-1);
850062        }
850063      //
850064      // Set the device number if necessary.
850065      //
850066      if (S_ISBLK (mode) || S_ISCHR (mode))
850067        {
850068          inode->direct[0] = device;
850069          inode->changed = 1;
850070        }
850071      //
850072      // Put the inode.
850073      //
850074      inode_put (inode);
850075      //
850076      // Return.
850077      //
850078      return (0);
850079  }
```

100.4.43 kernel/fs/path_mount.c

Si veda la sezione 99.3.41.

```
860001  #include <kernel/fs.h>
860002  #include <errno.h>
860003  #include <kernel/proc.h>
860004  //----------------------------------------------------------------------
860005  int
860006  path_mount (pid_t pid, const char *path_dev, const char *path_mnt,
860007              int options)
860008  {
860009      proc_t   *ps;
860010      dev_t     device;        // Device to mount.
860011      inode_t  *inode_mnt;     // Directory mount point.
860012      void     *pstatus;
860013      //
860014      // Get process.
860015      //
860016      ps = proc_reference (pid);
860017      //
860018      // Verify to be the super user.
```

```
860019      //
860020      if (ps->euid != 0)
860021        {
860022          errset (EPERM);          // Operation not permitted.
860023          return (-1);
860024        }
860025      //
860026      device = path_device (pid, path_dev);
860027      if (device < 0)
860028        {
860029          return (-1);
860030        }
860031      //
860032      inode_mnt = path_inode (pid, path_mnt);
860033      if (inode_mnt == NULL)
860034        {
860035          return (-1);
860036        }
860037      if (!S_ISDIR (inode_mnt->mode))
860038        {
860039          inode_put (inode_mnt);
860040          errset (ENOTDIR);        // Not a directory.
860041          return (-1);
860042        }
860043      if (inode_mnt->sb_attached != NULL)
860044        {
860045          inode_put (inode_mnt);
860046          errset (EBUSY);          // Device or resource busy.
860047          return (-1);
860048        }
860049      //
860050      // All data is available.
860051      //
860052      pstatus = sb_mount (device, &inode_mnt, options);
860053      if (pstatus == NULL)
860054        {
860055          inode_put (inode_mnt);
860056          return (-1);
860057        }
860058      //
860059      return (0);
860060  }
```

100.4.44 kernel/fs/path_stat.c

Si veda la sezione 99.3.50.

```
870001 #include <kernel/fs.h>
870002 #include <errno.h>
870003 #include <kernel/proc.h>
870004 //----------------------------------------------------------------------
870005 int
870006 path_stat (pid_t pid, const char *path, struct stat *buffer)
870007 {
870008     proc_t   *ps;
870009     inode_t  *inode;
870010     //
870011     // Get process.
870012     //
870013     ps = proc_reference (pid);
870014     //
870015     // Try to load the file inode.
870016     //
870017     inode = path_inode (pid, path);
870018     if (inode == NULL)
870019       {
870020         //
870021         // Cannot access the file: it does not exists or permissions are
870022         // not sufficient. Variable 'errno' is set by function
870023         // 'path_inode()'.
870024         //
870025         errset (errno);
870026         return (-1);
870027       }
870028     //
870029     // Inode loaded: update the buffer.
870030     //
870031     buffer->st_dev      = inode->sb->device;
870032     buffer->st_ino      = inode->ino;
870033     buffer->st_mode     = inode->mode;
870034     buffer->st_nlink    = inode->links;
870035     buffer->st_uid      = inode->uid;
870036     buffer->st_gid      = inode->gid;
870037     if (S_ISBLK (buffer->st_mode) || S_ISCHR (buffer->st_mode))
870038       {
870039         buffer->st_rdev = inode->direct[0];
870040       }
870041     else
870042       {
870043         buffer->st_rdev = 0;
870044       }
870045     buffer->st_size     = inode->size;
```

```
870046     buffer->st_atime    = inode->time;  // All times are the same for
870047     buffer->st_mtime    = inode->time;  // Minix 1 file system.
870048     buffer->st_ctime    = inode->time;  //
870049     buffer->st_blksize  = inode->sb->blksize;
870050     buffer->st_blocks   = inode->blkcnt;
870051     //
870052     // If the inode is a device special file, the 'st_rdev' value is
870053     // taken from the first direct zone (as of Minix 1 organization).
870054     //
870055     if (S_ISBLK(inode->mode) || S_ISCHR(inode->mode))
870056       {
870057         buffer->st_rdev = inode->direct[0];
870058       }
870059     else
870060       {
870061         buffer->st_rdev = 0;
870062       }
870063     //
870064     // Release the inode and return.
870065     //
870066     inode_put (inode);
870067     //
870068     // Return.
870069     //
870070     return (0);
870071 }
```

100.4.45 kernel/fs/path_umount.c

Si veda la sezione 99.3.41.

```
880001 #include <kernel/fs.h>
880002 #include <errno.h>
880003 #include <kernel/proc.h>
880004 //----------------------------------------------------------------------
880005 int
880006 path_umount (pid_t pid, const char *path_mnt)
880007 {
880008     proc_t   *ps;
880009     dev_t     device;                // Device to mount.
880010     inode_t  *inode_mount_point;     // Original mount point.
880011     inode_t  *inode;                 // Inode table.
880012     int       i;                     // Inode table index.
880013     //
880014     // Get process.
880015     //
880016     ps = proc_reference (pid);
880017     //
```

```
880018      // Verify to be the super user.
880019      //
880020      if (ps->euid != 0)
880021        {
880022          errset (EPERM);          // Operation not permitted.
880023          return (-1);
880024        }
880025      //
880026      // Get the directory mount point.
880027      //
880028      inode_mount_point = path_inode (pid, path_mnt);
880029      if (inode_mount_point == NULL)
880030        {
880031          errset (ENOENT);                  // No such file or directory.
880032          return (-1);
880033        }
880034      //
880035      // Verify that the path is a directory.
880036      //
880037      if (!S_ISDIR (inode_mount_point->mode))
880038        {
880039          inode_put (inode_mount_point);
880040          errset (ENOTDIR);                 // Not a directory.
880041          return (-1);
880042        }
880043      //
880044      // Verify that there is something attached.
880045      //
880046      device = inode_mount_point->sb_attached->device;
880047      if (device == 0)
880048        {
880049          //
880050          // There is nothing to unmount.
880051          //
880052          inode_put (inode_mount_point);
880053          errset (E_NOT_MOUNTED);           // Not mounted.
880054          return (-1);
880055        }
880056      //
880057      // Are there exactly two internal references? Let's explain:
880058      // the directory that act as mount point, should have one reference
880059      // because it is mounting something and another because it was just
880060      // opened again, a few lines above. If there are more references
880061      // it is wrong; if there are less, it is also wrong at this point.
880062      //
880063      if (inode_mount_point->references != 2)
880064        {
880065          inode_put (inode_mount_point);
```

```
880066            errset (EUNKNOWN);              // Unknown error.
880067            return (-1);
880068          }
880069        //
880070        // All data is available: find if there are open file inside
880071        // the file system to unmount. But first load the inode table
880072        // pointer.
880073        //
880074        inode = inode_reference ((dev_t) 0, (ino_t) 0);
880075        if (inode == NULL)
880076          {
880077            //
880078            // This error should not happen.
880079            //
880080            inode_put (inode_mount_point);
880081            errset (EUNKNOWN);              // Unknown error.
880082            return (-1);
880083          }
880084        //
880085        // Scan the inode table.
880086        //
880087        for (i = 0; i < INODE_MAX_SLOTS; i++)
880088          {
880089            if (inode[i].sb == inode_mount_point->sb_attached  &&
880090                inode[i].references > 0)
880091              {
880092                //
880093                // At least one file is open inside the super block to
880094                // release: cannot unmount.
880095                //
880096                inode_put (inode_mount_point);
880097                errset (EBUSY);             // Device or resource busy.
880098                return (-1);
880099              }
880100          }
880101        //
880102        // Can unmount: save and remove the super block memory;
880103        // clear the mount point reference and put inode.
880104        //
880105        inode_mount_point->sb_attached->changed          = 1;
880106        sb_save (inode_mount_point->sb_attached);
880107        //
880108        inode_mount_point->sb_attached->device           = 0;
880109        inode_mount_point->sb_attached->inode_mounted_on = NULL;
880110        inode_mount_point->sb_attached->blksize          = 0;
880111        inode_mount_point->sb_attached->options          = 0;
880112        //
880113        inode_mount_point->sb_attached                   = NULL;
```

```
880114      inode_mount_point->references                    = 0;
880115      inode_put (inode_mount_point);
880116      //
880117      inode_put (inode_mount_point);
880118      //
880119      return (0);
880120  }
```

100.4.46 kernel/fs/path_unlink.c

Si veda la sezione 99.3.44.

```
890001  #include <kernel/fs.h>
890002  #include <errno.h>
890003  #include <kernel/proc.h>
890004  #include <libgen.h>
890005  #include <kernel/k_libc.h>
890006  //----------------------------------------------------------------------
890007  int
890008  path_unlink (pid_t pid, const char *path)
890009  {
890010      proc_t       *ps;
890011      inode_t      *inode_unlink;
890012      inode_t      *inode_directory;
890013      char          path_unlink[PATH_MAX];
890014      char          path_copy[PATH_MAX];
890015      char         *path_directory;
890016      char         *name_unlink;
890017      dev_t         device;
890018      off_t         start;
890019      char          buffer[SB_MAX_ZONE_SIZE];
890020      directory_t *dir = (directory_t *) buffer;
890021      int           status;
890022      ssize_t       size_read;
890023      ssize_t       size_written;
890024      int           d;                      // Directory buffer index.
890025      //
890026      // Get process.
890027      //
890028      ps = proc_reference (pid);
890029      //
890030      // Get full paths.
890031      //
890032      path_full (path, ps->path_cwd, path_unlink);
890033      strncpy (path_copy, path_unlink, PATH_MAX);
890034      path_directory = dirname (path_copy);
890035      //
890036      // Get the inode to be unlinked.
```

```
890037      //
890038      inode_unlink = path_inode (pid, path_unlink);
890039      if (inode_unlink == NULL)
890040        {
890041          return (-1);
890042        }
890043      //
890044      // If it is a directory, verify that it is empty.
890045      //
890046      if (S_ISDIR (inode_unlink->mode))
890047        {
890048          if (!inode_dir_empty (inode_unlink))
890049            {
890050              inode_put (inode_unlink);
890051              errset (ENOTEMPTY);          // Directory not empty.
890052              return (-1);
890053            }
890054        }
890055      //
890056      // Get the inode of the directory containing it.
890057      //
890058      inode_directory = path_inode (pid, path_directory);
890059      if (inode_directory == NULL)
890060        {
890061          inode_put (inode_unlink);
890062          return (-1);
890063        }
890064      //
890065      // Check if something is mounted on the directory.
890066      //
890067      if (inode_directory->sb_attached != NULL)
890068        {
890069          //
890070          // Must select the right directory.
890071          //
890072          device = inode_directory->sb_attached->device;
890073          inode_put (inode_directory);
890074          inode_directory = inode_get (device, 1);
890075          if (inode_directory == NULL)
890076            {
890077              inode_put (inode_unlink);
890078              return (-1);
890079            }
890080        }
890081      //
890082      // Check if write is allowed for the file system.
890083      //
890084      if (inode_directory->sb->options & MOUNT_RO)
```

```
890085          {
890086            errset (EROFS);         // Read-only file system.
890087            return (-1);
890088          }
890089        //
890090        // Verify access permissions for the directory. The number "3" means
890091        // that the user must have access permission and write permission:
890092        // "-wx" == 2+1 == 3.
890093        //
890094        status = inode_check (inode_directory, S_IFDIR, 3, ps->uid);
890095        if (status != 0)
890096          {
890097            errset (EPERM);                 // Operation not permitted.
890098            inode_put (inode_unlink);
890099            inode_put (inode_directory);
890100            return (-1);
890101          }
890102        //
890103        // Get the base name to be unlinked: this will alter the
890104        // original path.
890105        //
890106        name_unlink = basename (path_unlink);
890107        //
890108        // Read the directory content and try to locate the item to unlink.
890109        //
890110        for (start  = 0;
890111             start <  inode_directory->size;
890112             start += inode_directory->sb->blksize)
890113          {
890114            size_read = inode_file_read (inode_directory, start, buffer,
890115                                         inode_directory->sb->blksize,
890116                                         NULL);
890117            if (size_read < sizeof (directory_t))
890118              {
890119                break;
890120              }
890121            //
890122            // Scan the directory portion just read, for the item to unlink.
890123            //
890124            dir = (directory_t *) buffer;
890125            //
890126            for (d = 0; d < size_read; d += (sizeof (directory_t)), dir++)
890127              {
890128                if (dir->ino != 0                                     &&
890129                    strncmp (dir->name, name_unlink, NAME_MAX) == 0)
890130                  {
890131                    //
890132                    // Found the corresponding item: unlink the inode.
```

```
890133                    //
890134                    dir->ino     = 0;
890135                    //
890136                    // Update the directory inside the file system.
890137                    //
890138                    size_written = inode_file_write (inode_directory, start,
890139                                                     buffer, size_read);
890140                    if (size_written != size_read)
890141                      {
890142                        //
890143                        // Write problem: just tell.
890144                        //
890145                        k_printf ("kernel alert: directory write error!\n");
890146                      }
890147                    //
890148                    // Update directory inode and put inode. If the unlinked
890149                    // inode was a directory, the parent directory inode
890150                    // must reduce the file system link count.
890151                    //
890152                    if (S_ISDIR (inode_unlink->mode))
890153                      {
890154                        inode_directory->links--;
890155                      }
890156                    inode_directory->time = k_time (NULL);
890157                    inode_directory->changed = 1;
890158                    inode_put (inode_directory);
890159                    //
890160                    // Reduce link inside unlinked inode and put inode.
890161                    //
890162                    inode_unlink->links--;
890163                    inode_unlink->changed = 1;
890164                    inode_unlink->time = k_time (NULL);
890165                    inode_put (inode_unlink);
890166                    //
890167                    // Just return, as the work is done.
890168                    //
890169                    return (0);
890170                  }
890171              }
890172          }
890173        //
890174        // At this point, it was not possible to unlink the file.
890175        //
890176        inode_put (inode_unlink);
890177        inode_put (inode_directory);
890178        errset (EUNKNOWN);            // Unknown error.
890179        return (-1);
```

```
890180 }
```

100.4.47 kernel/fs/sb_inode_status.c

Si veda la sezione 99.3.45.

```
900001 #include <kernel/fs.h>
900002 #include <errno.h>
900003 //----------------------------------------------------------------------
900004 int
900005 sb_inode_status (sb_t *sb, ino_t ino)
900006 {
900007     int map_element;
900008     int map_bit;
900009     int map_mask;
900010     //
900011     // Check arguments.
900012     //
900013     if (ino == 0 || sb == NULL)
900014       {
900015         errset (EINVAL);         // Invalid argument.
900016         return (-1);
900017       }
900018     //
900019     //  Calculate the map element, the map bit and the map mask.
900020     //
900021     map_element = ino / 16;
900022     map_bit     = ino % 16;
900023     map_mask    = 1 << map_bit;
900024     //
900025     // Check the inode and return.
900026     //
900027     if (sb->map_inode[map_element] & map_mask)
900028       {
900029         return (1);      // True.
900030       }
900031     else
900032       {
900033         return (0);      // False.
900034       }
900035 }
```

100.4.48 kernel/fs/sb_mount.c

«

Si veda la sezione 99.3.46.

```
910001 #include <kernel/fs.h>
910002 #include <errno.h>
910003 #include <kernel/devices.h>
910004 //----------------------------------------------------------------------
910005 sb_t *
910006 sb_mount (dev_t device, inode_t **inode_mnt, int options)
910007 {
910008     sb_t   *sb;
910009     ssize_t size_read;
910010     addr_t  start;
910011     int     m;
910012     size_t  size_sb;
910013     size_t  size_map;
910014     //
910015     // Find if it is already mounted.
910016     //
910017     sb = sb_reference (device);
910018     if (sb != NULL)
910019       {
910020         errset (EBUSY);           // Device or resource busy: device
910021         return (NULL);            // already mounted.
910022       }
910023     //
910024     // Find if `*inode_mnt' is already mounting something.
910025     //
910026     if (*inode_mnt != NULL && (*inode_mnt)->sb_attached != NULL)
910027       {
910028         errset (EBUSY);           // Device or resource busy: mount point
910029         return (NULL);            // already used.
910030       }
910031     //
910032     // The inode is not yet mounting anything, or it is new: find a free
910033     // slot inside the super block table.
910034     //
910035     sb = sb_reference ((dev_t) -1);
910036     if (sb == NULL)
910037       {
910038         errset (EBUSY);           // Device or resource busy:
910039         return (NULL);            // no free slots.
910040       }
910041     //
910042     // A free slot was found: the super block header must be loaded, but
910043     // before it is necessary to calculate the header size to be read.
910044     //
910045     size_sb = offsetof (sb_t, device);
```

```
910046      //
910047      // Then fix the starting point.
910048      //
910049      start  = 1024;                      // After boot block.
910050      //
910051      // Read the file system super block header.
910052      //
910053      size_read = dev_io ((pid_t) -1, device, DEV_READ, start, sb,
910054                          size_sb, NULL);
910055      if (size_read != size_sb)
910056        {
910057          errset (EIO);                   // I/O error.
910058          return (NULL);
910059        }
910060      //
910061      // Save some more data.
910062      //
910063      sb->device            =  device;
910064      sb->options           =  options;
910065      sb->inode_mounted_on = *inode_mnt;
910066      sb->blksize           =  (1024 << sb->log2_size_zone);
910067      //
910068      // Check if the super block data is valid.
910069      //
910070      if (sb->magic_number != 0x137F)
910071        {
910072          errset (ENODEV);        // No such device: unsupported
910073          sb->device = 0;         // file system type.
910074          return (NULL);
910075        }
910076      if (sb->map_inode_blocks > SB_MAX_INODE_BLOCKS)
910077        {
910078          errset (E_MAP_INODE_TOO_BIG);
910079          return (NULL);
910080        }
910081      if (sb->map_zone_blocks > SB_MAX_ZONE_BLOCKS)
910082        {
910083          errset (E_MAP_ZONE_TOO_BIG);
910084          return (NULL);
910085        }
910086      if (sb->blksize > SB_MAX_ZONE_SIZE)
910087        {
910088          errset (E_DATA_ZONE_TOO_BIG);
910089          return (NULL);
910090        }
910091      //
910092      // A right super block header was loaded from disk, now load the
910093      // super block inode bit map.
```

```
910094      //
910095      start  = 1024;                            // After boot block.
910096      start += 1024;                            // After super block.
910097      for (m = 0; m < SB_MAP_INODE_SIZE; m++)   //
910098        {                                       // Reset map in memory,
910099          sb->map_inode[m] = 0xFFFF;            // before loading.
910100        }                                       //
910101      size_map = sb->map_inode_blocks * 1024;
910102      size_read = dev_io ((pid_t) -1, sb->device, DEV_READ, start,
910103                          sb->map_inode, size_map, NULL);
910104      if (size_read != size_map)
910105        {
910106          errset (EIO);                 // I/O error.
910107          return (NULL);
910108        }
910109      //
910110      // Load the super block zone bit map.
910111      //
910112      start  = 1024;                            // After boot block.
910113      start += 1024;                            // After super block.
910114      start += (sb->map_inode_blocks * 1024);   // After inode bit map.
910115      for (m = 0; m < SB_MAP_ZONE_SIZE; m++)    //
910116        {                                       // Reset map in memory,
910117          sb->map_zone[m] = 0xFFFF;             // before loading.
910118        }                                       //
910119      size_map = sb->map_zone_blocks * 1024;
910120      size_read = dev_io ((pid_t) -1, sb->device, DEV_READ, start,
910121                          sb->map_zone, size_map, NULL);
910122      if (size_read != size_map)
910123        {
910124          errset (EIO);                 // I/O error.
910125          return (NULL);
910126        }
910127      //
910128      // Check the inode that should mount the super block. If
910129      // `*inode_mnt' is `NULL', then it is meant to be the first mount of
910130      // the root file system. In such case, the inode must be loaded too,
910131      // and the value for `*inode_mnt' must be modified.
910132      //
910133      if (*inode_mnt == NULL)
910134        {
910135          *inode_mnt = inode_get (device, 1);
910136        }
910137      //
910138      // Check for a valid value.
910139      //
910140      if (*inode_mnt == NULL)
910141        {
```

```
910142             //
910143             // This is bad!
910144             //
910145             errset (EUNKNOWN);                // Unknown error.
910146             return (NULL);
910147           }
910148       //
910149       // A valid inode is available for the mount.
910150       //
910151       (*inode_mnt)->sb_attached  = sb;
910152       //
910153       // Return the super block pointer.
910154       //
910155       return (sb);
910156   }
```

100.4.49 kernel/fs/sb_reference.c

Si veda la sezione 99.3.47.

```
920001   #include <kernel/fs.h>
920002   #include <errno.h>
920003   //----------------------------------------------------------------------
920004   sb_t *
920005   sb_reference (dev_t device)
920006   {
920007       int s;                                // Slot index.
920008       //
920009       // If device is zero, a reference to the whole table is returned.
920010       //
920011       if (device == 0)
920012         {
920013           return (sb_table);
920014         }
920015       //
920016       // If device is ((dev_t) -1), a reference to a free slot is
920017       // returned.
920018       //
920019       if (device == ((dev_t) -1))
920020         {
920021           for (s = 0; s < SB_MAX_SLOTS; s++)
920022             {
920023               if (sb_table[s].device == 0)
920024                 {
920025                   return (&sb_table[s]);
920026                 }
920027             }
920028           return (NULL);
```

```
920029          }
920030      //
920031      // A device was selected: find the super block associated to it.
920032      //
920033      for (s = 0; s < SB_MAX_SLOTS; s++)
920034        {
920035          if (sb_table[s].device == device)
920036            {
920037              return (&sb_table[s]);
920038            }
920039        }
920040      //
920041      // The super block was not found.
920042      //
920043      return (NULL);
920044  }
```

100.4.50 kernel/fs/sb_save.c

Si veda la sezione 99.3.48.

```
930001  #include <kernel/fs.h>
930002  #include <errno.h>
930003  #include <kernel/devices.h>
930004  //----------------------------------------------------------------------
930005  int
930006  sb_save (sb_t *sb)
930007  {
930008      ssize_t size_written;
930009      addr_t  start;
930010      size_t  size_map;
930011      //
930012      // Check for valid argument.
930013      //
930014      if (sb == NULL)
930015        {
930016          errset (EINVAL);                        // Invalid argument.
930017          return (-1);
930018        }
930019      //
930020      // Check if the super block changed for some reason (only the
930021      // inode and the zone maps can change really).
930022      //
930023      if (!sb->changed)
930024        {
930025          //
930026          // Nothing to save.
930027          //
```

```
930028          return (0);
930029        }
930030      //
930031      // Something inside the super block changed: start the procedure to
930032      // save the inode map (recall that the super block header is not
930033      // saved, because it never changes).
930034      //
930035      start  = 1024;                              // After boot block.
930036      start += 1024;                              // After super block.
930037      size_map = sb->map_inode_blocks * 1024;
930038      size_written = dev_io ((pid_t) -1, sb->device, DEV_WRITE, start,
930039                  sb->map_inode, size_map, NULL);
930040      if (size_written != size_map)
930041        {
930042          //
930043          // Error writing the map.
930044          //
930045          errset (EIO);                    // I/O error.
930046          return (-1);
930047        }
930048      //
930049      // Start the procedure to save the zone map.
930050      //
930051      start  = 1024;                              // After boot block.
930052      start += 1024;                              // After super block.
930053      start += (sb->map_inode_blocks * 1024);     // After inode bit map.
930054      size_map = sb->map_zone_blocks * 1024;
930055      size_written = dev_io ((pid_t) -1, sb->device, DEV_WRITE, start,
930056                  sb->map_zone, size_map, NULL);
930057      if (size_written != size_map)
930058        {
930059          //
930060          // Error writing the map.
930061          //
930062          errset (EIO);                    // I/O error.
930063          return (-1);
930064        }
930065      //
930066      // Super block saved.
930067      //
930068      sb->changed = 0;
930069      //
930070      return (0);
930071  }
```

100.4.51 kernel/fs/sb_table.c

Si veda la sezione 99.3.47.

```
940001 #include <kernel/fs.h>
940002 //----------------------------------------------------------------------
940003 sb_t    sb_table[SB_MAX_SLOTS];
```

100.4.52 kernel/fs/sb_zone_status.c

Si veda la sezione 99.3.45.

```
950001 #include <kernel/fs.h>
950002 #include <errno.h>
950003 //----------------------------------------------------------------------
950004 int
950005 sb_zone_status (sb_t *sb, zno_t zone)
950006 {
950007     int map_element;
950008     int map_bit;
950009     int map_mask;
950010     //
950011     // Check arguments.
950012     //
950013     if (zone == 0 || sb == NULL)
950014       {
950015         errset (EINVAL);        // Invalid argument.
950016         return (-1);
950017       }
950018     //
950019     //  Calculate the map element, the map bit and the map mask.
950020     //
950021     map_element = zone / 16;
950022     map_bit     = zone % 16;
950023     map_mask    = 1 << map_bit;
950024     //
950025     // Check the zone and return.
950026     //
950027     if (sb->map_zone[map_element] & map_mask)
950028       {
950029         return (1);     // True.
950030       }
950031     else
950032       {
950033         return (0);     // False.
950034       }
950035 }
```

100.4.53 kernel/fs/zone_alloc.c

«

Si veda la sezione 99.3.51.

```
960001 #include <kernel/fs.h>
960002 #include <kernel/devices.h>
960003 #include <errno.h>
960004 //----------------------------------------------------------------------
960005 zno_t
960006 zone_alloc (sb_t *sb)
960007 {
960008     int    m;                       // Index inside the inode map.
960009     int    map_element;
960010     int    map_bit;
960011     int    map_mask;
960012     zno_t  zone;
960013     char   buffer[SB_MAX_ZONE_SIZE];
960014     int    status;
960015     //
960016     // Verify if write is allowed.
960017     //
960018     if (sb->options & MOUNT_RO)
960019       {
960020         errset (EROFS);            // Read-only file system.
960021         return ((zno_t) 0);
960022       }
960023     //
960024     // Write allowed: scan the zone map, to find a free zone.
960025     // If a free zone can be found, allocate it inside the map.
960026     // Index 'm' starts from one, because the first bit of the
960027     // map is reserved for a 'zero' data-zone that does not
960028     // exist: the second bit is for the real first data-zone.
960029     //
960030     for (zone = 0, m = 1; m < (SB_MAP_ZONE_SIZE * 16); m++)
960031       {
960032         map_element = m / 16;
960033         map_bit     = m % 16;
960034         map_mask    = 1 << map_bit;
960035         if (!(sb->map_zone[map_element] & map_mask))
960036           {
960037             //
960038             // Found a free place: set the map.
960039             //
960040             sb->map_zone[map_element] |= map_mask;
960041             sb->changed = 1;
960042             //
960043             // The *second* bit inside the map is for the first data
960044             // zone (the zone after the inode table inside the file
960045             // system), because the first is for a special 'zero' data
```

```
960046                  // zone, not really used.
960047                  //
960048                  zone = sb->first_data_zone + m - 1; // Found a free zone.
960049                  //
960050                  // If the zone is outside the disk size, let set the map
960051                  // bit, but reset variable 'zone'.
960052                  //
960053                  if (zone >= sb->zones)
960054                    {
960055                      zone = 0;
960056                    }
960057                  else
960058                    {
960059                      break;
960060                    }
960061                }
960062            }
960063          if (zone == 0)
960064            {
960065              errset (ENOSPC);        // No space left on device.
960066              return ((zno_t) 0);
960067            }
960068          //
960069          // A free zone was found and the map was modified inside
960070          // the super block in memory. The zone must be cleared.
960071          //
960072          status = zone_write (sb, zone, buffer);
960073          if (status != 0)
960074            {
960075              zone_free (sb, zone);
960076              return ((zno_t) 0);
960077            }
960078          //
960079          // A zone was allocated: return the number.
960080          //
960081          return (zone);
960082  }
```

100.4.54 kernel/fs/zone_free.c

«

Si veda la sezione 99.3.51.

```
970001  #include <kernel/fs.h>
970002  #include <kernel/devices.h>
970003  #include <errno.h>
970004  //----------------------------------------------------------------------
970005  int
970006  zone_free (sb_t *sb, zno_t zone)
```

```
970007 {
970008     int map_element;
970009     int map_bit;
970010     int map_mask;
970011     //
970012     // Check arguments.
970013     //
970014     if (sb == NULL || zone < sb->first_data_zone)
970015       {
970016         errset (EINVAL);        // Invalid argument.
970017         return (-1);
970018       }
970019     //
970020     // Calculate the map element, the map bit and the map mask.
970021     //
970022     // The *second* bit inside the map is for the first data-zone
970023     // (the zone after the inode table inside the file system),
970024     // because the first is for a special 'zero' data-zone, not
970025     // really used.
970026     //
970027     map_element = (zone - sb->first_data_zone + 1) / 16;
970028     map_bit     = (zone - sb->first_data_zone + 1) % 16;
970029     map_mask    = 1 << map_bit;
970030     //
970031     // Verify if the requested zone is inside the file system area.
970032     //
970033     if (zone >= sb->zones)
970034       {
970035         errset (EINVAL);        // Invalid argument.
970036         return (-1);
970037       }
970038     //
970039     // Free the zone and return.
970040     //
970041     if (sb->map_zone[map_element] & map_mask)
970042       {
970043         sb->map_zone[map_element] &= ~map_mask;
970044         sb->changed = 1;
970045         return (0);
970046       }
970047     else
970048       {
970049         errset (EUNKNOWN);              // The zone was already free.
970050         return (-1);
970051       }
970052 }
```

100.4.55 kernel/fs/zone_read.c

«

Si veda la sezione 99.3.53.

```
980001 #include <sys/os16.h>
980002 #include <kernel/fs.h>
980003 #include <kernel/devices.h>
980004 #include <errno.h>
980005 //----------------------------------------------------------------------
980006 int
980007 zone_read (sb_t *sb, zno_t zone, void *buffer)
980008 {
980009     size_t   size_zone;
980010     off_t    off_start;
980011     ssize_t  size_read;
980012     //
980013     // Verify if the requested zone is inside the file system area.
980014     //
980015     if (zone >= sb->zones)
980016       {
980017         errset (EINVAL);        // Invalid argument.
980018         return (-1);
980019       }
980020     //
980021     // Calculate start position.
980022     //
980023     size_zone  = 1024 << sb->log2_size_zone;
980024     off_start  = zone;
980025     off_start *= size_zone;
980026     //
980027     // Read from device to the buffer.
980028     //
980029     size_read = dev_io ((pid_t) -1, sb->device, DEV_READ, off_start,
980030                         buffer, size_zone, NULL);
980031     if (size_read != size_zone)
980032       {
980033         errset (EIO);           // I/O error.
980034         return (-1);
980035       }
980036     else
980037       {
980038         return (0);
980039       }
980040 }
```

100.4.56 kernel/fs/zone_write.c

Si veda la sezione 99.3.53.

```
990001 #include <kernel/fs.h>
990002 #include <kernel/devices.h>
990003 #include <errno.h>
990004 //----------------------------------------------------------------------
990005 int
990006 zone_write (sb_t *sb, zno_t zone, void *buffer)
990007 {
990008     size_t   size_zone;
990009     off_t    off_start;
990010     ssize_t  size_written;
990011     //
990012     // Verify if write is allowed.
990013     //
990014     if (sb->options & MOUNT_RO)
990015       {
990016         errset (EROFS);          // Read-only file system.
990017         return (-1);
990018       }
990019     //
990020     // Verify if the requested zone is inside the file system area.
990021     //
990022     if (zone >= sb->zones)
990023       {
990024         errset (EINVAL);         // Invalid argument.
990025         return (-1);
990026       }
990027     //
990028     // Write is allowed: calculate start position.
990029     //
990030     size_zone  = 1024 << sb->log2_size_zone;
990031     off_start  = zone;
990032     off_start *= size_zone;
990033     //
990034     // Write the buffer to the device.
990035     //
990036     size_written = dev_io ((pid_t) -1, sb->device, DEV_WRITE, off_start,
990037                            buffer, size_zone, NULL);
990038     if (size_written != size_zone)
990039       {
990040         errset (EIO);            // I/O error.
990041         return (-1);
990042       }
990043     else
990044       {
990045         return (0);
```

```
990046          }
990047  }
```

100.5 os16: «kernel/ibm_i86.h»

Si veda la sezione 99.4.

```
1000001 #ifndef _KERNEL_IBM_I86_H
1000002 #define _KERNEL_IBM_I86_H 1
1000003 
1000004 #include <stdint.h>
1000005 #include <size_t.h>
1000006 #include <kernel/memory.h>
1000007 #include <sys/types.h>
1000008 //----------------------------------------------------------------------
1000009 #define IBM_I86_VIDEO_MODE   0x02
1000010 #define IBM_I86_VIDEO_PAGES  4
1000011 
1000012 #define IBM_I86_VIDEO_COLUMNS   80
1000013 #define IBM_I86_VIDEO_ROWS      25
1000014 #define IBM_I86_VIDEO_ADDRESS   0xB8000L, 0xB9000L, 0xBA000L, 0xBB000L
1000015 //----------------------------------------------------------------------
1000016 void     _int10_00 (uint16_t video_mode);
1000017 void     _int10_02 (uint16_t page, uint16_t position);
1000018 void     _int10_05 (uint16_t page);
1000019 uint16_t _int12    (void);
1000020 uint16_t _int13_00 (uint16_t drive);
1000021 uint16_t _int13_02 (uint16_t drive, uint16_t sectors,
1000022                     uint16_t cylinder, uint16_t head,
1000023                     uint16_t sector, void *buffer);
1000024 uint16_t _int13_03 (uint16_t drive, uint16_t sectors,
1000025                     uint16_t cylinder, uint16_t head,
1000026                     uint16_t sector, void *buffer);
1000027 uint16_t _int16_00 (void);
1000028 uint16_t _int16_01 (void);
1000029 uint16_t _int16_02 (void);
1000030 
1000031 #define int10_00(video_mode)      (_int10_00 ((uint16_t) video_mode))
1000032 #define int10_02(page, position) (_int10_02 ((uint16_t) page, \
1000033                                             (uint16_t) position))
1000034 #define int10_05(page)            (_int10_05 ((uint16_t) page))
1000035 #define int12()                   ((unsigned int) _int12 ())
1000036 
1000037 #define int13_00(drive)           ((unsigned int) \
1000038                                    _int13_00 ((uint16_t) drive))
1000039 #define int13_02(drive, sectors, cylinder, head, sector, buffer) \
1000040                                   ((unsigned int) \
1000041                                    _int13_02 ((uint16_t) drive, \
```

```
1000042                                                 (uint16_t) sectors, \
1000043                                                 (uint16_t) cylinder, \
1000044                                                 (uint16_t) head, \
1000045                                                 (uint16_t) sector, \
1000046                                                 buffer))
1000047 #define int13_03(drive, sectors, cylinder, head, sector, buffer) \
1000048                                   ((unsigned int) \
1000049                                    _int13_03 ((uint16_t) drive, \
1000050                                                 (uint16_t) sectors, \
1000051                                                 (uint16_t) cylinder, \
1000052                                                 (uint16_t) head, \
1000053                                                 (uint16_t) sector, \
1000054                                                 buffer))
1000055 #define int16_00()                ((unsigned int) _int16_00 ())
1000056 #define int16_01()                ((unsigned int) _int16_01 ())
1000057 #define int16_02()                ((unsigned int) _int16_02 ())
1000058 //----------------------------------------------------------------------
1000059 uint16_t _in_8   (uint16_t port);
1000060 uint16_t _in_16  (uint16_t port);
1000061 void     _out_8  (uint16_t port, uint16_t value);
1000062 void     _out_16 (uint16_t port, uint16_t value);
1000063
1000064 #define in_8(port)            ((unsigned int) _in_8 ((uint16_t) port))
1000065 #define in_16(port)           ((unsigned int) _in_16 ((uint16_t) port))
1000066 #define out_8(port, value)    (_out_8 ((uint16_t) port, \
1000067                                        (uint16_t) value))
1000068 #define out_16(port, value)   (_out_16 ((uint16_t) port, \
1000069                                         (uint16_t) value))
1000070 //----------------------------------------------------------------------
1000071 void _cli     (void);
1000072 void _sti     (void);
1000073
1000074 #define cli() (_cli ())
1000075 #define sti() (_sti ())
1000076 //----------------------------------------------------------------------
1000077 void irq_on  (unsigned int irq);
1000078 void irq_off (unsigned int irq);
1000079 //----------------------------------------------------------------------
1000080 void     _ram_copy      (segment_t org_seg, offset_t org_off,
1000081                          segment_t dst_seg, offset_t dst_off,
1000082                          uint16_t size);
1000083
1000084 #define ram_copy(org_seg, org_off, dst_seg, dst_off, size) \
1000085                         (_ram_copy ((uint16_t) org_seg, \
1000086                                     (uint16_t) org_off, \
1000087                                     (uint16_t) dst_seg, \
1000088                                     (uint16_t) dst_off, \
1000089                                     (uint16_t) size))
```

```
1000090 //----------------------------------------------------------------------
1000091 void con_select     (int console);
1000092 void con_putc       (int console, int c);
1000093 void con_scroll     (int console);
1000094 int  con_char_wait  (void);
1000095 int  con_char_read  (void);
1000096 int  con_char_ready (void);
1000097 void con_init       (void);
1000098 //----------------------------------------------------------------------
1000099 #define DSK_MAX                   4
1000100 #define DSK_SECTOR_SIZE         512     // Fixed!
1000101 
1000102 typedef struct {
1000103     unsigned int  bios_drive;
1000104     unsigned int  cylinders;
1000105     unsigned int  heads;
1000106     unsigned int  sectors;
1000107     unsigned int  retry;
1000108 } dsk_t;
1000109 
1000110 typedef struct {
1000111     unsigned int  cylinder;
1000112     unsigned int  head;
1000113     unsigned int  sector;
1000114 } dsk_chs_t;
1000115 //----------------------------------------------------------------------
1000116 extern dsk_t dsk_table[DSK_MAX];
1000117 //----------------------------------------------------------------------
1000118 void   dsk_setup          (void);
1000119 int    dsk_reset          (int drive);
1000120 void   dsk_sector_to_chs (int drive, unsigned int sector,
1000121                            dsk_chs_t *chs);
1000122 int    dsk_read_sectors  (int drive, unsigned int start_sector,
1000123                            void *buffer, unsigned int n_sectors);
1000124 int    dsk_write_sectors (int drive, unsigned int start_sector,
1000125                            void *buffer, unsigned int n_sectors);
1000126 size_t dsk_read_bytes    (int drive, off_t offset,
1000127                            void *buffer, size_t count);
1000128 size_t dsk_write_bytes   (int drive, off_t offset,
1000129                            void *buffer, size_t count);
1000130 //----------------------------------------------------------------------
1000131 
1000132 #endif
```

100.5.1 kernel/ibm_i86/_cli.s

«

Si veda la sezione 99.4.

```
1010001 .global __cli
1010002 ;------------------------------------------------------------------------
1010003 .text
1010004 ;------------------------------------------------------------------------
1010005 ; Clear interrupt flag.
1010006 ;------------------------------------------------------------------------
1010007 .align 2
1010008 __cli:
1010009     cli
1010010     ret
```

100.5.2 kernel/ibm_i86/_in_16.s

«

Si veda la sezione 99.4.

```
1020001 .global __in_16
1020002 ;------------------------------------------------------------------------
1020003 .text
1020004 ;------------------------------------------------------------------------
1020005 ; Port input word.
1020006 ;------------------------------------------------------------------------
1020007 __in_16:
1020008     enter #2, #0        ; 1 local variable.
1020009     pushf
1020010     cli
1020011     pusha
1020012     mov   dx, 4[bp]     ; 1st arg (port number).
1020013     in    ax, dx
1020014     mov   -2[bp], ax    ; Save AX.
1020015     popa
1020016     popf
1020017     mov   ax, -2[bp]    ; AX is the function return value.
1020018     leave
1020019     ret
```

100.5.3 kernel/ibm_i86/_in_8.s

«

Si veda la sezione 99.4.

```
1030001 .global __in_8
1030002 ;------------------------------------------------------------------------
1030003 .text
1030004 ;------------------------------------------------------------------------
1030005 ; Port input byte.
```

```
1030006 ;----------------------------------------------------------------------
1030007 __in_8:
1030008     enter #2, #0        ; 1 local variable.
1030009     pushf
1030010     cli
1030011     pusha
1030012     mov   dx, 4[bp]     ; 1st arg (port number).
1030013     in    al, dx
1030014     mov   ah, #0
1030015     mov   -2[bp], ax    ; Save AX.
1030016     popa
1030017     popf
1030018     mov   ax, -2[bp]    ; AX is the function return value.
1030019     leave
1030020     ret
```

100.5.4 kernel/ibm_i86/_int10_00.s

Si veda la sezione 99.4.

```
1040001 .global __int10_00
1040002 ;----------------------------------------------------------------------
1040003 .text
1040004 ;----------------------------------------------------------------------
1040005 ; INT 0x10 - video - set video mode
1040006 ;        AH = 0x00
1040007 ;        AL = desired video mode:
1040008 ;             0x00 = text 40x25  pages 8
1040009 ;             0x01 = text 40x25  pages 8
1040010 ;             0x02 = text 80x25  pages 4
1040011 ;             0x03 = text 80x25  pages 4
1040012 ;             0x07 = text 80x25  pages 4?
1040013 ;
1040014 ; Specify the display mode for the currently active display adapter.
1040015 ;----------------------------------------------------------------------
1040016 .align 2
1040017 __int10_00:
1040018     enter #0, #0        ; No local variables.
1040019     pushf
1040020     cli
1040021     pusha
1040022     mov   ah, #0x00
1040023     mov   al, 4[bp]     ; 1st arg (video mode).
1040024     int   #0x10
1040025     popa
1040026     popf
1040027     leave
1040028     ret
```

100.5.5 kernel/ibm_i86/_int10_02.s

«

Si veda la sezione 99.4.

```
1050001 .global __int10_02
1050002 ;----------------------------------------------------------------------
1050003 .text
1050004 ;----------------------------------------------------------------------
1050005 ; INT 0x10 - video - set cursor position
1050006 ;       AH = 0x02
1050007 ;       BH = page number:
1050008 ;            0-7 in modes 0 and 1
1050009 ;            0-3 in modes 2 and 3
1050010 ;       DH = row (0x00 is top)
1050011 ;       DL = column (0x00 is left)
1050012 ;----------------------------------------------------------------------
1050013 .align 2
1050014 __int10_02:
1050015     enter #0, #0        ; No local variables.
1050016     pushf
1050017     cli
1050018     pusha
1050019     mov   ah, #0x02
1050020     mov   bh, #0x00
1050021     mov   bh, 4[bp]     ; 1st arg (page).
1050022     mov   dx, 6[bp]     ; 2nd arg (pos).
1050023     int   #0x10
1050024     popa
1050025     popf
1050026     leave
1050027     ret
```

100.5.6 kernel/ibm_i86/_int10_05.s

«

Si veda la sezione 99.4.

```
1060001 .global __int10_05
1060002 ;----------------------------------------------------------------------
1060003 .text
1060004 ;----------------------------------------------------------------------
1060005 ; INT 0x10 - video - select active display page
1060006 ;       AH = 0x05
1060007 ;       AL = new page number (0x00 is the first)
1060008 ;----------------------------------------------------------------------
1060009 .align 2
1060010 __int10_05:
1060011     enter #0, #0        ; No local variables.
1060012     pushf
1060013     cli
```

```
1060014     pusha
1060015     mov   ah, #0x05
1060016     mov   bh, 4[bp]      ; 1st arg (page).
1060017     int   #0x10
1060018     popa
1060019     popf
1060020     leave
1060021     ret
```

100.5.7 kernel/ibm_i86/_int12.s

Si veda la sezione 99.4.

```
1070001 .global __int12
1070002 ;-----------------------------------------------------------------------
1070003 .text
1070004 ;-----------------------------------------------------------------------
1070005 ; INT 12 - bios - get memory size
1070006 ; Return:
1070007 ;       AX = kilobytes of contiguous memory starting at absolute address
1070008 ;            0x00000
1070009 ;
1070010 ; This call returns the contents of the word at absolute address
1070011 ; 0x00413.
1070012 ;-----------------------------------------------------------------------
1070013 .align 2
1070014 __int12:
1070015     enter #2, #0         ; 1 local variable.
1070016     pushf
1070017     cli
1070018     pusha
1070019     int   #0x12
1070020     mov   -2[bp], ax     ; save AX.
1070021     popa
1070022     popf
1070023     mov   ax, -2[bp]     ; AX is the function return value.
1070024     leave
1070025     ret
```

100.5.8 kernel/ibm_i86/_int13_00.s

Si veda la sezione 99.4.

```
1080001 .global __int13_00
1080002 ;-----------------------------------------------------------------------
1080003 .text
1080004 ;-----------------------------------------------------------------------
```

```
1080005 ; INT 0x13 - disk - reset disk system
1080006 ;        AH = 0x00
1080007 ;        DL = drive (if bit 7 is set both hard disks and floppy disks
1080008 ;             reset)
1080009 ; Return:
1080010 ;        AH = status
1080011 ;        CF clear if successful (returned AH=0x00)
1080012 ;        CF set on error
1080013 ;-----------------------------------------------------------------------
1080014 .align 2
1080015 __int13_00:
1080016     enter #2, #0        ; 1 local variable.
1080017     pushf
1080018     cli
1080019     pusha
1080020     mov   ah, #0x00
1080021     mov   dl, 4[bp]     ; 1st arg.
1080022     int   #0x13
1080023     mov   al, #0x00
1080024     mov   -2[bp], ax    ; save AX.
1080025     popa
1080026     popf
1080027     mov   ax, -2[bp]    ; AX is the function return value.
1080028     leave
1080029     ret
```

100.5.9 kernel/ibm_i86/_int13_02.s

Si veda la sezione 99.4.

```
1090001 .global __int13_02
1090002 ;-----------------------------------------------------------------------
1090003 .text
1090004 ;-----------------------------------------------------------------------
1090005 ; INT 0x13 - disk - read sectors into memory
1090006 ;        AH = 0x02
1090007 ;        AL = number of sectors to read (must be nonzero)
1090008 ;        CH = cylinder number (0-255)
1090009 ;        CL bit 6-7 =
1090010 ;             cylinder number (256-1023)
1090011 ;        CL bit 0-5 =
1090012 ;             sector number   (1-63)
1090013 ;        DH = head number     (0-255)
1090014 ;        DL = drive number (bit 7 set for hard disk)
1090015 ;        ES:BX -> data buffer
1090016 ; Return:
1090017 ;        CF set on error
1090018 ;        CF clear if successful
```

```
1090019 ;        AH = status (0x00 if successful)
1090020 ;        AL = number of sectors transferred (only valid if CF set for
1090021 ;             some BIOSes)
1090022 ;----------------------------------------------------------------------
1090023 .align 2
1090024 __int13_02:
1090025     enter #2, #0        ; 1 local variable.
1090026     pushf
1090027     cli
1090028     pusha
1090029     mov   ax, ds        ; Set ES the same as DS.
1090030     mov   es, ax        ;
1090031     mov   ax,  8[bp]    ; 3rd arg (cylinder). It must be splitted and
1090032     mov   ch, al        ; assigned to the right registers.
1090033     mov   cl, ah        ;
1090034     shl   cl, 1         ;
1090035     shl   cl, 1         ;
1090036     shl   cl, 1         ;
1090037     shl   cl, 1         ;
1090038     shl   cl, 1         ;
1090039     shl   cl, 1         ;
1090040     add   cl, 12[bp]    ; 5th arg (sector).
1090041     mov   dl,  4[bp]    ; 1st arg (drive).
1090042     mov   al,  6[bp]    ; 2nd arg (sectors to be read).
1090043     mov   dh, 10[bp]    ; 4th arg (head).
1090044     mov   bx, 14[bp]    ; 6th arg (buffer pointer).
1090045     mov   ah, #0x02
1090046     int   #0x13
1090047     mov   -2[bp], ax    ; save AX.
1090048     popa
1090049     popf
1090050     mov   ax, -2[bp]    ; AX is the function return value.
1090051     leave
1090052     ret
```

100.5.10 kernel/ibm_i86/_int13_03.s

«

Si veda la sezione 99.4.

```
1100001 .global __int13_03
1100002 ;----------------------------------------------------------------------
1100003 .text
1100004 ;----------------------------------------------------------------------
1100005 ; INT 0x13 - disk - write sectors to disk
1100006 ;        AH = 0x03
1100007 ;        AL = number of sectors to write (must be nonzero)
1100008 ;        CH = cylinder number (0-255)
1100009 ;        CL bit 6-7 =
```

```
1100010 ;             cylinder number (256-1023)
1100011 ;        CL bit 0-5 =
1100012 ;             sector number   (1-63)
1100013 ;        DH = head number     (0-255)
1100014 ;        DL = drive number (bit 7 set for hard disk)
1100015 ;        ES:BX -> data buffer
1100016 ; Return:
1100017 ;        CF set on error
1100018 ;        CF clear if successful
1100019 ;        AH = status (0x00 if successful)
1100020 ;        AL = number of sectors transferred (only valid if CF set for
1100021 ;             some BIOSes)
1100022 ;----------------------------------------------------------------------
1100023 .align 2
1100024 __int13_03:
1100025     enter #2, #0        ; 1 local variable.
1100026     pushf
1100027     cli
1100028     pusha
1100029     mov   ax, ds        ; Set ES the same as DS.
1100030     mov   es, ax        ;
1100031     mov   ax,  8[bp]    ; 3rd arg (cylinder). It must be splitted and
1100032     mov   ch, al        ; assigned to the right registers.
1100033     mov   cl, ah        ;
1100034     shl   cl, 1         ;
1100035     shl   cl, 1         ;
1100036     shl   cl, 1         ;
1100037     shl   cl, 1         ;
1100038     shl   cl, 1         ;
1100039     shl   cl, 1         ;
1100040     add   cl, 12[bp]    ; 5th arg (sector).
1100041     mov   dl,  4[bp]    ; 1st arg (drive).
1100042     mov   al,  6[bp]    ; 2nd arg (sectors to be written).
1100043     mov   dh, 10[bp]    ; 4th arg (head).
1100044     mov   bx, 14[bp]    ; 6th arg (buffer pointer).
1100045     mov   ah, #0x03
1100046     int   #0x13
1100047     mov   -2[bp], ax    ; save AX.
1100048     popa
1100049     popf
1100050     mov   ax, -2[bp]    ; AX is the function return value.
1100051     leave
1100052     ret
```

100.5.11 kernel/ibm_i86/_int16_00.s

«

Si veda la sezione 99.4.

```
1110001 .global __int16_00
1110002 ;----------------------------------------------------------------------
1110003 .text
1110004 ;----------------------------------------------------------------------
1110005 ; INT 0x16 - keyboard - get keystroke
1110006 ;       AH = 0x00
1110007 ; Return:
1110008 ;       AH = BIOS scan code
1110009 ;       AL = ASCII character
1110010 ;----------------------------------------------------------------------
1110011 .align 2
1110012 __int16_00:
1110013     enter #2, #0        ; 1 local variable.
1110014     pushf
1110015     cli
1110016     pusha
1110017     mov   ah, #0x00
1110018     int   #0x16
1110019     mov   -2[bp], ax    ; Save AX.
1110020     popa
1110021     popf
1110022     mov   ax, -2[bp]    ; AX is the function return value.
1110023     leave
1110024     ret
```

100.5.12 kernel/ibm_i86/_int16_01.s

«

Si veda la sezione 99.4.

```
1120001 .global __int16_01
1120002 ;----------------------------------------------------------------------
1120003 .text
1120004 ;----------------------------------------------------------------------
1120005 ; INT 0x16 - keyboard - check for keystroke
1120006 ;       AH = 0x01
1120007 ; Return:
1120008 ;       ZF set if no keystroke available
1120009 ;       ZF clear if keystroke available
1120010 ;           AH = BIOS scan code
1120011 ;           AL = ASCII character
1120012 ;
1120013 ; If a keystroke is present, it is not removed from the keyboard buffer.
1120014 ;----------------------------------------------------------------------
1120015 .align 2
1120016 __int16_01:
```

```
1120017       enter #2, #0         ; 1 local variable.
1120018       pushf
1120019       cli
1120020       pusha
1120021       mov   ah, #0x01
1120022       int   #0x16
1120023       jnz   __int16_01_ok
1120024       mov   ax, #0         ; Put zero to AX, if no keystroke is available.
1120025   __int16_01_ok:
1120026       mov   -2[bp], ax     ; Save AX.
1120027       popa
1120028       popf
1120029       mov   ax, -2[bp]     ; AX is the function return value.
1120030       leave
1120031       ret
```

100.5.13 kernel/ibm_i86/_int16_02.s

«

Si veda la sezione 99.4.

```
1130001   .global __int16_02
1130002   ;----------------------------------------------------------------------
1130003   .text
1130004   ;----------------------------------------------------------------------
1130005   ; INT 0x16 - keyboard - get shift flags
1130006   ;       AH = 0x02
1130007   ; Return:
1130008   ;       AL = shift flags
1130009   ;       AH might be destroyed
1130010   ;
1130011   ; bit 7    Insert active
1130012   ; bit 6    CapsLock active
1130013   ; bit 5    NumLock active
1130014   ; bit 4    ScrollLock active
1130015   ; bit 3    Alt key pressed
1130016   ; bit 2    Ctrl key pressed
1130017   ; bit 1    left shift key pressed
1130018   ; bit 0    right shift key pressed
1130019   ;----------------------------------------------------------------------
1130020   .align 2
1130021   __int16_02:
1130022       enter #2, #0         ; 1 local variable.
1130023       pushf
1130024       cli
1130025       pusha
1130026       mov   ah, #0x02
1130027       int   #0x16
1130028       mov   ah, #0         ; Reset AH.
```

```
1130029       mov    -2[bp], ax     ; Save AX.
1130030       popa
1130031       popf
1130032       mov    ax, -2[bp]     ; AX is the function return value.
1130033       leave
1130034       ret
```

100.5.14 kernel/ibm_i86/_out_16.s

«

Si veda la sezione 99.4.

```
1140001 .global __out_16
1140002 ;-----------------------------------------------------------------------
1140003 .text
1140004 ;-----------------------------------------------------------------------
1140005 ; Port output word.
1140006 ;-----------------------------------------------------------------------
1140007 .align 2
1140008 __out_16:
1140009     enter #0, #0          ; No local variables.
1140010     pushf
1140011     cli
1140012     pusha
1140013     mov    dx, 4[bp]      ; 1st arg (port number).
1140014     mov    ax, 6[bp]      ; 2nd arg (value).
1140015     out    dx, ax
1140016     popa
1140017     popf
1140018     leave
1140019     ret
```

100.5.15 kernel/ibm_i86/_out_8.s

«

Si veda la sezione 99.4.

```
1150001 .global __out_8
1150002 ;-----------------------------------------------------------------------
1150003 .text
1150004 ;-----------------------------------------------------------------------
1150005 ; Port output byte.
1150006 ;-----------------------------------------------------------------------
1150007 .align 2
1150008 __out_8:
1150009     enter #0, #0          ; No local variables.
1150010     pushf
1150011     cli
1150012     pusha
```

```
1150013     mov   dx, 4[bp]       ; 1st arg (port number).
1150014     mov   ax, 6[bp]       ; 2nd arg (value).
1150015     out   dx, al
1150016     popa
1150017     popf
1150018     leave
1150019     ret
```

100.5.16 kernel/ibm_i86/_ram_copy.s

Si veda la sezione 99.4.

```
1160001 .global __ram_copy
1160002 ;----------------------------------------------------------------------
1160003 .text
1160004 ;----------------------------------------------------------------------
1160005 ; Copy some bytes between segments.
1160006 ;----------------------------------------------------------------------
1160007 .align 2
1160008 __ram_copy:
1160009     enter #0, #0          ; No local variables.
1160010     pushf
1160011     cli
1160012     pusha
1160013     mov   ax,  4[bp]      ; 1st arg (source segment).
1160014     mov   si,  6[bp]      ; 2nd arg (source offset).
1160015     mov   bx,  8[bp]      ; 3rd arg (destination segment).
1160016     mov   di, 10[bp]      ; 4th arg (destination offset).
1160017     mov   cx, 12[bp]      ; 5th arg (size).
1160018     mov   dx, ds          ; save the data segment.
1160019     mov   ds, ax          ; set DS.
1160020     mov   es, bx          ; set ES.
1160021     rep
1160022     movsb                 ; Copy the array of bytes.
1160023     mov   ds, dx          ; Restore the data segment.
1160024     mov   es, dx          ; Restore or fix the extra segment.
1160025     popa
1160026     popf
1160027     leave
1160028     ret
```

100.5.17 kernel/ibm_i86/_sti.s

«

Si veda la sezione 99.4.

```
1170001 .global __sti
1170002 ;------------------------------------------------------------------------
1170003 .text
1170004 ;------------------------------------------------------------------------
1170005 ; Set interrupt flag.
1170006 ;------------------------------------------------------------------------
1170007 .align 2
1170008 __sti:
1170009     sti
1170010     ret
```

100.5.18 kernel/ibm_i86/con_char_read.c

«

Si veda la sezione 99.4.

```
1180001 #include <kernel/ibm_i86.h>
1180002 #include <kernel/k_libc.h>
1180003 #include <sys/os16.h>
1180004 #include <sys/types.h>
1180005 #include <stdint.h>
1180006 //----------------------------------------------------------------------
1180007 int
1180008 con_char_read (void)
1180009 {
1180010     int c;
1180011     c = int16_01 ();
1180012     //
1180013     // Remove special keys that are not used: they have zero in the low
1180014     // 8 bits, and something in the upper 8 bits.
1180015     //
1180016     if ((c & 0xFF00) && !(c & 0x00FF))
1180017       {
1180018         int16_00 ();     // Remove from buffer and return zero:
1180019         return (0);      // no key.
1180020       }
1180021     //
1180022     // A common key was pressed: filter only che low 8 bits.
1180023     //
1180024     c = c & 0x00FF;
1180025     if (c == 0)
1180026       {
1180027         return (c);      // There is no key.
1180028       }
1180029     if (c == '\r')       // Convert 'CR' to 'LF'.
1180030       {
```

```
1180031             c = '\n';
1180032           }
1180033         int16_00 ();           // Remove the key from buffer and return.
1180034         return (c);
1180035     }
```

100.5.19 kernel/ibm_i86/con_char_ready.c

Si veda la sezione 99.4.

```
1190001     #include <kernel/ibm_i86.h>
1190002     #include <kernel/k_libc.h>
1190003     #include <sys/os16.h>
1190004     #include <sys/types.h>
1190005     #include <stdint.h>
1190006     //----------------------------------------------------------------------
1190007     int
1190008     con_char_ready (void)
1190009     {
1190010         int c;
1190011         c = int16_01 ();
1190012         //
1190013         // Remove special keys that are not used: they have zero in the low
1190014         // 8 bits, and something in the upper 8 bits.
1190015         //
1190016         if ((c & 0xFF00) && !(c & 0x00FF))
1190017           {
1190018             int16_00 ();      // Remove from buffer and return zero:
1190019             return (0);       // no key.
1190020           }
1190021         //
1190022         // A common key was pressed: filter only che low 8 bits.
1190023         //
1190024         c = c & 0x00FF;
1190025         return (c);
1190026     }
```

100.5.20 kernel/ibm_i86/con_char_wait.c

Si veda la sezione 99.4.

```
1200001     #include <kernel/ibm_i86.h>
1200002     #include <kernel/k_libc.h>
1200003     #include <sys/os16.h>
1200004     #include <sys/types.h>
1200005     #include <stdint.h>
1200006     //----------------------------------------------------------------------
```

```
1200007 int
1200008 con_char_wait (void)
1200009 {
1200010     int c;
1200011     c = int16_00 ();
1200012     c = c & 0x00FF;
1200013     if (c == '\r')
1200014       {
1200015         c = '\n';
1200016       }
1200017     return (c);
1200018 }
```

100.5.21 kernel/ibm_i86/con_init.c

Si veda la sezione 99.4.

```
1210001 #include <kernel/ibm_i86.h>
1210002 #include <kernel/k_libc.h>
1210003 #include <sys/os16.h>
1210004 #include <sys/types.h>
1210005 #include <stdint.h>
1210006 //----------------------------------------------------------------------
1210007 void
1210008 con_init (void)
1210009 {
1210010     int page;
1210011     //
1210012     int10_00 (IBM_I86_VIDEO_MODE);
1210013     int10_05 (0);
1210014     //
1210015     for (page = 0; page < IBM_I86_VIDEO_PAGES; page++)
1210016       {
1210017         con_putc (page, '\n');
1210018       }
1210019 }
```

100.5.22 kernel/ibm_i86/con_putc.c

Si veda la sezione 99.4.

```
1220001 #include <kernel/ibm_i86.h>
1220002 #include <kernel/k_libc.h>
1220003 #include <sys/os16.h>
1220004 #include <sys/types.h>
1220005 #include <stdint.h>
1220006 //----------------------------------------------------------------------
```

```
1220007 void
1220008 con_putc (int console, int c)
1220009 {
1220010     static int      cursor[IBM_I86_VIDEO_PAGES];
1220011     static addr_t   address[] = {IBM_I86_VIDEO_ADDRESS};
1220012            addr_t   address_destination;
1220013            size_t   size_screen;
1220014            size_t   size_row;
1220015            uint16_t cell;
1220016            uint16_t attribute = 0x0700;
1220017            int      cursor_row;
1220018            int      cursor_column;
1220019            int      cursor_combined;
1220020
1220021     if (console < 0 || console >= IBM_I86_VIDEO_PAGES)
1220022       {
1220023         //
1220024         // No such console.
1220025         //
1220026         return;
1220027       }
1220028     //
1220029     // Calculate sizes.
1220030     //
1220031     size_row    = IBM_I86_VIDEO_COLUMNS;
1220032     size_screen = size_row * IBM_I86_VIDEO_ROWS;
1220033     //
1220034     // See if it is a special character, or if the cursor position
1220035     // requires a scroll up.
1220036     //
1220037     if (c == '\n')
1220038       {
1220039         con_scroll (console);
1220040         cursor[console] = (size_screen - size_row);
1220041       }
1220042     else if (c == '\b')
1220043       {
1220044         cursor[console]--;
1220045         if (cursor[console] < 0)
1220046           {
1220047             cursor[console] = 0;
1220048           }
1220049       }
1220050     else if (cursor[console] == (size_screen -1))
1220051       {
1220052         //
1220053         // Scroll up.
1220054         //
```

```
1220055            con_scroll (console);
1220056            //
1220057            cursor[console] -= size_row;
1220058          }
1220059        //
1220060        // If it is not a control character, print it.
1220061        //
1220062        if (c != '\n' && c != '\b')
1220063          {
1220064            //
1220065            // Write the character.
1220066            //
1220067            address_destination  = address[console];
1220068            address_destination += (cursor[console] * 2);
1220069            cell = (attribute | (c & 0x00FF));
1220070            //
1220071            mem_write (address_destination, &cell, sizeof (uint16_t));
1220072            //
1220073            // and an extra space after it (to be able to show the cursor).
1220074            //
1220075            cell = (attribute | ' ');
1220076            address_destination  += 2;
1220077            mem_write (address_destination, &cell, sizeof (uint16_t));
1220078            //
1220079            //
1220080            //
1220081            cursor[console]++;
1220082          }
1220083        //
1220084        // Update the cursor position on screen.
1220085        //
1220086        cursor_row      = cursor[console] / size_row;
1220087        cursor_column   = cursor[console] % size_row;
1220088        cursor_combined = (cursor_row << 8) | cursor_column;
1220089        //
1220090        // Set cursor position.
1220091        //
1220092        int10_02 (console, cursor_combined);
1220093    }
```

100.5.23 kernel/ibm_i86/con_scroll.c

«

Si veda la sezione 99.4.

```
1230001  #include <kernel/ibm_i86.h>
1230002  #include <kernel/k_libc.h>
1230003  #include <sys/os16.h>
1230004  #include <sys/types.h>
```

```
1230005 #include <stdint.h>
1230006 //----------------------------------------------------------------------
1230007 void
1230008 con_scroll (int console)
1230009 {
1230010     static addr_t   address[] = {IBM_I86_VIDEO_ADDRESS};
1230011            addr_t   address_source;
1230012            addr_t   address_destination;
1230013            size_t   size_screen;
1230014            size_t   size_row;
1230015     static uint16_t empty_line[IBM_I86_VIDEO_COLUMNS];
1230016     //
1230017     size_row    = IBM_I86_VIDEO_COLUMNS;
1230018     size_screen = size_row * IBM_I86_VIDEO_ROWS;
1230019     //
1230020     // Scroll up.
1230021     //
1230022     address_source       = address[console];
1230023     address_source      += size_row * 2;
1230024     address_destination = address[console];
1230025     //
1230026     mem_copy (address_source, address_destination,
1230027               (size_t) ((size_screen - size_row) * 2));
1230028     //
1230029     address_destination  = address[console];
1230030     address_destination += ((size_screen - size_row) * 2);
1230031     //
1230032     mem_write (address_destination, &empty_line,
1230033                (size_t) (size_row * 2));
1230034 }
```

100.5.24 kernel/ibm_i86/con_select.c

Si veda la sezione 99.4.

```
1240001 #include <kernel/ibm_i86.h>
1240002 #include <kernel/k_libc.h>
1240003 #include <sys/os16.h>
1240004 #include <sys/types.h>
1240005 #include <stdint.h>
1240006 //----------------------------------------------------------------------
1240007 void
1240008 con_select (int console)
1240009 {
1240010     //
1240011     // Variable 'console' gose from zero to 'IBM_I86_VIDEO_PAGES - 1'.
1240012     //
1240013     if (console >= 0 && console < IBM_I86_VIDEO_PAGES)
```

```
1240014         {
1240015           int10_05 (console);
1240016         }
1240017 }
```

100.5.25 kernel/ibm_i86/dsk_read_bytes.c

«

Si veda la sezione 99.4.

```
1250001 #include <kernel/ibm_i86.h>
1250002 #include <kernel/k_libc.h>
1250003 #include <sys/os16.h>
1250004 #include <sys/types.h>
1250005 #include <stdint.h>
1250006 //----------------------------------------------------------------------
1250007 size_t
1250008 dsk_read_bytes (int drive, off_t offset, void *buffer, size_t count)
1250009 {
1250010     unsigned char *data_buffer = (unsigned char *) buffer;
1250011     int              status;
1250012     unsigned int     sector;
1250013     unsigned char    sector_buffer[DSK_SECTOR_SIZE];
1250014     int              i;
1250015     int              j = 0;
1250016     size_t           k = 0;
1250017 
1250018     sector = offset / DSK_SECTOR_SIZE;
1250019     i      = offset % DSK_SECTOR_SIZE;
1250020 
1250021     status = dsk_read_sectors (drive, sector, sector_buffer, 1);
1250022 
1250023     if (status != 0)
1250024       {
1250025         return ((size_t) 0);
1250026       }
1250027 
1250028     while (count)
1250029       {
1250030         for (; i < DSK_SECTOR_SIZE && count > 0;
1250031                i++, j++, k++, count--, offset++)
1250032           {
1250033             data_buffer[j] = sector_buffer[i];
1250034           }
1250035         if (count)
1250036           {
1250037             sector = offset / DSK_SECTOR_SIZE;
1250038             i      = offset % DSK_SECTOR_SIZE;
1250039             status = dsk_read_sectors (drive, sector, sector_buffer, 1);
```

```
1250040                if (status != 0)
1250041                  {
1250042                    return (k);
1250043                  }
1250044              }
1250045          }
1250046        return (k);
1250047    }
```

100.5.26 kernel/ibm_i86/dsk_read_sectors.c

Si veda la sezione 99.4.

```
1260001 #include <kernel/ibm_i86.h>
1260002 #include <kernel/k_libc.h>
1260003 #include <sys/os16.h>
1260004 #include <sys/types.h>
1260005 #include <stdint.h>
1260006 //----------------------------------------------------------------------
1260007 int
1260008 dsk_read_sectors (int drive, unsigned int start_sector, void *buffer,
1260009                   unsigned int n_sectors)
1260010 {
1260011     int          status;
1260012     unsigned int retry;
1260013     unsigned int remaining;
1260014     dsk_chs_t    chs;
1260015     dsk_sector_to_chs (drive, start_sector, &chs);
1260016     remaining = dsk_table[drive].sectors - chs.sector + 1;
1260017     if (remaining < n_sectors)
1260018       {
1260019         status = dsk_read_sectors (drive, start_sector, buffer,
1260020                                    remaining);
1260021         if (status == 0)
1260022           {
1260023             status = dsk_read_sectors (drive, start_sector + remaining,
1260024                                        buffer, n_sectors - remaining);
1260025           }
1260026         return (status);
1260027       }
1260028     else
1260029       {
1260030         for (retry = 0; retry < dsk_table[drive].retry; retry++)
1260031           {
1260032             status = int13_02 (dsk_table[drive].bios_drive, n_sectors,
1260033                                chs.cylinder, chs.head, chs.sector,
1260034                                buffer);
1260035             status = status & 0x00F0;
```

```
1260036                 if (status == 0)
1260037                   {
1260038                     break;
1260039                   }
1260040                 else
1260041                   {
1260042                     dsk_reset (drive);
1260043                   }
1260044               }
1260045           }
1260046         if (status == 0)
1260047           {
1260048             return (0);
1260049           }
1260050         else
1260051           {
1260052             return (-1);
1260053           }
1260054 }
```

100.5.27 kernel/ibm_i86/dsk_reset.c

«

Si veda la sezione 99.4.

```
1270001 #include <kernel/ibm_i86.h>
1270002 #include <kernel/k_libc.h>
1270003 #include <sys/os16.h>
1270004 #include <sys/types.h>
1270005 #include <stdint.h>
1270006 //----------------------------------------------------------------------
1270007 int
1270008 dsk_reset (int drive)
1270009 {
1270010     unsigned int status;
1270011     status = int13_00 (dsk_table[drive].bios_drive);
1270012     if (status == 0)
1270013       {
1270014         return (0);
1270015       }
1270016     else
1270017       {
1270018         return (-1);
1270019       }
1270020 }
```

100.5.28 kernel/ibm_i86/dsk_sector_to_chs.c

Si veda la sezione 99.4.

```
1280001 #include <kernel/ibm_i86.h>
1280002 #include <kernel/k_libc.h>
1280003 #include <sys/os16.h>
1280004 #include <sys/types.h>
1280005 #include <stdint.h>
1280006 //----------------------------------------------------------------------
1280007 void
1280008 dsk_sector_to_chs (int drive, unsigned int sector, dsk_chs_t *chs)
1280009 {
1280010     unsigned int sectors_per_cylinder;
1280011     sectors_per_cylinder = (dsk_table[drive].sectors
1280012                             * dsk_table[drive].heads);
1280013     chs->cylinder        = sector / sectors_per_cylinder;
1280014     sector               = sector % sectors_per_cylinder;
1280015     chs->head            = sector / dsk_table[drive].sectors;
1280016     sector               = sector % dsk_table[drive].sectors;
1280017     chs->sector          = sector + 1;
1280018 }
```

100.5.29 kernel/ibm_i86/dsk_setup.c

Si veda la sezione 99.4.

```
1290001 #include <kernel/ibm_i86.h>
1290002 //----------------------------------------------------------------------
1290003 void
1290004 dsk_setup (void)
1290005 {
1290006   dsk_reset (0);
1290007   dsk_table[0].bios_drive = 0x00;  // A: 1440 Kibyte floppy disk.
1290008   dsk_table[0].cylinders  = 80;
1290009   dsk_table[0].heads      = 2;
1290010   dsk_table[0].sectors    = 18;
1290011   dsk_table[0].retry      = 3;
1290012   dsk_reset (1);
1290013   dsk_table[1].bios_drive = 0x01;  // B: 1440 Kibyte floppy disk.
1290014   dsk_table[1].cylinders  = 80;
1290015   dsk_table[1].heads      = 2;
1290016   dsk_table[1].sectors    = 18;
1290017   dsk_table[1].retry      = 3;
1290018   dsk_reset (2);
1290019   dsk_table[2].bios_drive = 0x80;  // C: like a 2880 Kibyte floppy disk.
1290020   dsk_table[2].cylinders  = 80;
1290021   dsk_table[2].heads      = 2;
1290022   dsk_table[2].sectors    = 36;
```

```
1290023     dsk_table[2].retry       = 3;
1290024     dsk_reset (3);
1290025     dsk_table[3].bios_drive = 0x81;  // D: like a 2880 Kibyte floppy disk.
1290026     dsk_table[3].cylinders  = 80;
1290027     dsk_table[3].heads      = 2;
1290028     dsk_table[3].sectors    = 36;
1290029     dsk_table[3].retry      = 3;
1290030 }
```

100.5.30 kernel/ibm_i86/dsk_table.c

Si veda la sezione 99.4.

```
1300001 #include <kernel/ibm_i86.h>
1300002 //----------------------------------------------------------------------
1300003 dsk_t dsk_table[DSK_MAX];
```

100.5.31 kernel/ibm_i86/dsk_write_bytes.c

Si veda la sezione 99.4.

```
1310001 #include <kernel/ibm_i86.h>
1310002 #include <kernel/k_libc.h>
1310003 #include <sys/os16.h>
1310004 #include <sys/types.h>
1310005 #include <stdint.h>
1310006 //----------------------------------------------------------------------
1310007 size_t
1310008 dsk_write_bytes (int drive, off_t offset, void *buffer, size_t count)
1310009 {
1310010     unsigned char *data_buffer = (unsigned char *) buffer;
1310011     int             status;
1310012     unsigned int    sector;
1310013     unsigned char   sector_buffer[DSK_SECTOR_SIZE];
1310014     int             i;
1310015     int             j = 0;
1310016     size_t          k = 0;
1310017     size_t          m = 0;
1310018 
1310019     sector = offset / DSK_SECTOR_SIZE;
1310020     i      = offset % DSK_SECTOR_SIZE;
1310021     status = dsk_read_sectors (drive, sector, sector_buffer, 1);
1310022 
1310023     if (status != 0)
1310024       {
1310025         return ((size_t) 0);
1310026       }
```

```
1310027
1310028     while (count)
1310029       {
1310030         m = k;
1310031         for (; i < DSK_SECTOR_SIZE && count > 0;
1310032                i++, j++, k++, count--, offset++)
1310033           {
1310034             sector_buffer[i] = data_buffer[j];
1310035           }
1310036         status = dsk_write_sectors (drive, sector, sector_buffer, 1);
1310037         if (status != 0)
1310038           {
1310039             return (m);
1310040           }
1310041         if (count)
1310042           {
1310043             sector = offset / DSK_SECTOR_SIZE;
1310044             i      = offset % DSK_SECTOR_SIZE;
1310045             status = dsk_read_sectors (drive, sector, sector_buffer, 1);
1310046             if (status != 0)
1310047               {
1310048                 return (m);
1310049               }
1310050           }
1310051       }
1310052     return (k);
1310053 }
1310054
1310055
```

100.5.32 kernel/ibm_i86/dsk_write_sectors.c

Si veda la sezione 99.4.

```
1320001 #include <kernel/ibm_i86.h>
1320002 #include <kernel/k_libc.h>
1320003 #include <sys/os16.h>
1320004 #include <sys/types.h>
1320005 #include <stdint.h>
1320006 //----------------------------------------------------------------------
1320007 int
1320008 dsk_write_sectors (int drive, unsigned int start_sector, void *buffer,
1320009                    unsigned int n_sectors)
1320010 {
1320011     int          status;
1320012     unsigned int retry;
1320013     unsigned int remaining;
1320014     dsk_chs_t    chs;
```

```
1320015     dsk_sector_to_chs (drive, start_sector, &chs);
1320016     remaining = dsk_table[drive].sectors - chs.sector + 1;
1320017     if (remaining < n_sectors)
1320018       {
1320019         status = dsk_write_sectors (drive, start_sector,
1320020                                     buffer, remaining);
1320021         if (status == 0)
1320022           {
1320023             status = dsk_write_sectors (drive,
1320024                                         start_sector + remaining,
1320025                                         buffer, n_sectors - remaining);
1320026           }
1320027         return (status);
1320028       }
1320029     else
1320030       {
1320031         for (retry = 0; retry < dsk_table[drive].retry; retry++)
1320032           {
1320033             status = int13_03 (dsk_table[drive].bios_drive, n_sectors,
1320034                                chs.cylinder, chs.head, chs.sector,
1320035                                buffer);
1320036             status = status & 0x00F0;
1320037             if (status == 0)
1320038               {
1320039                 break;
1320040               }
1320041             else
1320042               {
1320043                 dsk_reset (drive);
1320044               }
1320045           }
1320046       }
1320047     if (status == 0)
1320048       {
1320049         return (0);
1320050       }
1320051     else
1320052       {
1320053         return (-1);
1320054       }
1320055 }
```

100.5.33 kernel/ibm_i86/irq_off.c

Si veda la sezione 99.4.

```
1330001 #include <kernel/ibm_i86.h>
1330002 #include <kernel/k_libc.h>
1330003 #include <sys/os16.h>
1330004 #include <sys/types.h>
1330005 #include <stdint.h>
1330006 //----------------------------------------------------------------------
1330007 void
1330008 irq_off (unsigned int irq)
1330009 {
1330010     unsigned int mask;
1330011     unsigned int status;
1330012     if (irq > 7)
1330013       {
1330014         return; // Only XT IRQs are handled.
1330015       }
1330016     else
1330017       {
1330018         mask   = (1 << irq);
1330019         status = in_8 (0x21);
1330020         status = status | mask;
1330021         out_8 (0x21, status);
1330022       }
1330023 }
```

100.5.34 kernel/ibm_i86/irq_on.c

Si veda la sezione 99.4.

```
1340001 #include <kernel/ibm_i86.h>
1340002 #include <kernel/k_libc.h>
1340003 #include <sys/os16.h>
1340004 #include <sys/types.h>
1340005 #include <stdint.h>
1340006 //----------------------------------------------------------------------
1340007 void
1340008 irq_on (unsigned int irq)
1340009 {
1340010     unsigned int mask;
1340011     unsigned int status;
1340012     if (irq > 7)
1340013       {
1340014         return; // Only XT IRQs are handled.
1340015       }
1340016     else
1340017       {
```

```
1340018         mask   = ~(1 << irq);
1340019         status = in_8 (0x21);
1340020         status = status & mask;
1340021         out_8 (0x21, status);
1340022       }
1340023 }
```

100.6 os16: «kernel/k_libc.h»

Si veda la sezione 99.5.

```
1350001 #ifndef _KERNEL_K_LIBC_H
1350002 #define _KERNEL_K_LIBC_H       1
1350003 
1350004 #include <const.h>
1350005 #include <restrict.h>
1350006 #include <size_t.h>
1350007 #include <clock_t.h>
1350008 #include <time_t.h>
1350009 #include <sys/types.h>
1350010 #include <stdarg.h>
1350011 
1350012 //----------------------------------------------------------------------
1350013 void    k_exit     (int status);
1350014 //----------------------------------------------------------------------
1350015 clock_t k_clock    (void);
1350016 int     k_stime    (time_t *timer);
1350017 time_t  k_time     (time_t *timer);
1350018 //----------------------------------------------------------------------
1350019 int     k_puts     (const char *string);
1350020 int     k_printf   (const char *restrict format, ...);
1350021 int     k_vprintf  (const char *restrict format, va_list arg);
1350022 int     k_vsprintf (char *restrict string, const char *restrict format,
1350023                     va_list arg);
1350024 void    k_perror   (const char *s);
1350025 
1350026 int     k_kill     (pid_t pid, int sig);
1350027 int     k_open     (const char *file, int oflags, ...);
1350028 void    k_close    (int fd);
1350029 ssize_t k_read     (int fd, void *buffer, size_t count);
1350030 //----------------------------------------------------------------------
1350031 
1350032 #endif
```

100.6.1 kernel/k_libc/k_clock.c

Si veda la sezione 99.5.

```
1360001 #include <kernel/k_libc.h>
1360002 //----------------------------------------------------------------------
1360003 extern clock_t _clock_ticks;     // uint32_t
1360004 //----------------------------------------------------------------------
1360005 clock_t
1360006 k_clock (void)
1360007 {
1360008     return (_clock_ticks);
1360009 }
```

100.6.2 kernel/k_libc/k_close.c

Si veda la sezione 99.5.

```
1370001 #include <kernel/k_libc.h>
1370002 #include <kernel/fs.h>
1370003 //----------------------------------------------------------------------
1370004 void
1370005 k_close (int fdn)
1370006 {
1370007     fd_close ((pid_t) 0, fdn);
1370008     return;
1370009 }
```

100.6.3 kernel/k_libc/k_exit.s

Si veda la sezione 99.5.

```
1380001 .global _k_exit
1380002 ;-----------------------------------------------------------------------
1380003 .text
1380004 ;-----------------------------------------------------------------------
1380005 .align 2
1380006 _k_exit:
1380007 halt:
1380008     hlt
1380009     jmp halt
```

100.6.4 kernel/k_libc/k_kill.c

«

Si veda la sezione 99.5.

```
1390001 #include <kernel/k_libc.h>
1390002 #include <kernel/proc.h>
1390003 //----------------------------------------------------------------------
1390004 int
1390005 k_kill (pid_t pid, int sig)
1390006 {
1390007     return (proc_sys_kill ((pid_t) 0, pid, sig));
1390008 }
```

100.6.5 kernel/k_libc/k_open.c

«

Si veda la sezione 99.5.

```
1400001 #include <kernel/k_libc.h>
1400002 #include <kernel/fs.h>
1400003 #include <stdarg.h>
1400004 #include <string.h>
1400005 #include <errno.h>
1400006 //----------------------------------------------------------------------
1400007 int
1400008 k_open (const char *file, int oflags, ...)
1400009 {
1400010     mode_t  mode;
1400011     va_list ap;
1400012     //
1400013     va_start (ap, oflags);
1400014     mode = va_arg (ap, mode_t);
1400015     //
1400016     if (file == NULL || strlen (file) == 0)
1400017       {
1400018         errset (EINVAL);          // Invalid argument.
1400019         return (-1);
1400020       }
1400021     return (fd_open ((pid_t) 0, file, oflags, mode));
1400022 }
```

100.6.6 kernel/k_libc/k_perror.c

«

Si veda la sezione 99.5.

```
1410001 #include <kernel/k_libc.h>
1410002 #include <errno.h>
1410003 //----------------------------------------------------------------------
1410004 void
```

```
1410005 k_perror (const char *s)
1410006 {
1410007     //
1410008     // If errno is zero, there is nothing to show.
1410009     //
1410010     if (errno == 0)
1410011       {
1410012         return;
1410013       }
1410014     //
1410015     // Show the string if there is one.
1410016     //
1410017     if (s != NULL && strlen (s) > 0)
1410018       {
1410019         k_printf ("%s: ", s);
1410020       }
1410021     //
1410022     // Show the translated error.
1410023     //
1410024     if (errfn[0] != 0 && errln != 0)
1410025       {
1410026         k_printf ("[%s:%u:%i] %s\n",
1410027                   errfn, errln, errno, strerror (errno));
1410028       }
1410029     else
1410030       {
1410031         k_printf ("[%i] %s\n", errno, strerror (errno));
1410032       }
1410033 }
```

100.6.7 kernel/k_libc/k_printf.c

Si veda la sezione 99.5.

```
1420001 #include <stdarg.h>
1420002 #include <kernel/k_libc.h>
1420003 //----------------------------------------------------------------------
1420004 int
1420005 k_printf  (const char *restrict format, ...)
1420006 {
1420007     va_list ap;
1420008     va_start (ap, format);
1420009     return k_vprintf (format, ap);
1420010 }
```

100.6.8 kernel/k_libc/k_puts.c

«

Si veda la sezione 99.5.

```
1430001 #include <sys/os16.h>
1430002 #include <kernel/devices.h>
1430003 #include <kernel/k_libc.h>
1430004 #include <string.h>
1430005 //----------------------------------------------------------------------
1430006 int
1430007 k_puts (const char *string)
1430008 {
1430009     dev_io ((pid_t) 0, DEV_TTY, DEV_WRITE, (off_t) 0, string,
1430010             strlen (string), NULL);
1430011     dev_io ((pid_t) 0, DEV_TTY, DEV_WRITE, (off_t) 0, "\n", 1, NULL);
1430012     return 1;
1430013 }
```

100.6.9 kernel/k_libc/k_read.c

«

Si veda la sezione 99.5.

```
1440001 #include <kernel/k_libc.h>
1440002 #include <kernel/fs.h>
1440003 //----------------------------------------------------------------------
1440004 ssize_t
1440005 k_read (int fdn, void *buffer, size_t count)
1440006 {
1440007     int     eof;
1440008     ssize_t size;
1440009     //
1440010     eof = 0;
1440011     //
1440012     while (1)
1440013       {
1440014         size += fd_read ((pid_t) 0, fdn, buffer, count, &eof);
1440015         if (size != 0 || eof)
1440016           {
1440017             break;
1440018           }
1440019       }
1440020     return (size);
1440021 }
```

100.6.10 kernel/k_libc/k_stime.c

Si veda la sezione 99.5.

```
1450001 #include <kernel/k_libc.h>
1450002 //----------------------------------------------------------------------
1450003 extern time_t  _clock_seconds;  // uint32_t
1450004 //----------------------------------------------------------------------
1450005 int
1450006 k_stime (time_t *timer)
1450007 {
1450008     _clock_seconds = *timer;
1450009     return (0);
1450010 }
```

100.6.11 kernel/k_libc/k_time.c

Si veda la sezione 99.5.

```
1460001 #include <kernel/k_libc.h>
1460002 #include <stddef.h>
1460003 //----------------------------------------------------------------------
1460004 extern time_t  _clock_seconds;  // uint32_t
1460005 //----------------------------------------------------------------------
1460006 time_t
1460007 k_time (time_t *timer)
1460008 {
1460009     if (timer != NULL)
1460010       {
1460011         *timer = _clock_seconds;
1460012       }
1460013     return (_clock_seconds);
1460014 }
```

100.6.12 kernel/k_libc/k_vprintf.c

Si veda la sezione 99.5.

```
1470001 #include <sys/os16.h>
1470002 #include <kernel/devices.h>
1470003 #include <stdarg.h>
1470004 #include <kernel/k_libc.h>
1470005 #include <string.h>
1470006 //----------------------------------------------------------------------
1470007 int
1470008 k_vprintf (const char *restrict format, va_list arg)
1470009 {
1470010     char  string[BUFSIZ];
```

```
1470011     int   ret;
1470012     string[0] = 0;
1470013     ret = k_vsprintf (string, format, arg);
1470014     dev_io ((pid_t) 0, DEV_CONSOLE, DEV_WRITE, (off_t) 0, string,
1470015             strlen (string), NULL);
1470016     return ret;
1470017 }
```

100.6.13 kernel/k_libc/k_vsprintf.c

«

Si veda la sezione 99.5.

```
1480001 #include <stdarg.h>
1480002 #include <kernel/k_libc.h>
1480003 #include <stdio.h>
1480004 //----------------------------------------------------------------------
1480005 int
1480006 k_vsprintf (char *restrict string, const char *restrict format,
1480007             va_list arg)
1480008 {
1480009     int   ret;
1480010     ret = vsnprintf (string, BUFSIZ, format, arg);
1480011     return ret;
1480012 }
```

100.7 os16: «kernel/main.h»

«

Si veda la sezione 99.6.

```
1490001 #ifndef _KERNEL_MAIN_H
1490002 #define _KERNEL_MAIN_H  1
1490003
1490004 #include <sys/types.h>
1490005
1490006 void  menu  (void);
1490007 pid_t run   (char *path, char *argv[], char *envp[]);
1490008 int   main (int argc, char *argv[], char *envp[]);
1490009
1490010 #endif
```

100.7.1 kernel/main/build.h

Si veda la sezione 99.6.

```
1500001 #define BUILD_DATE "2010.07.26 16:33:58"
```

100.7.2 kernel/main/crt0.s

Si veda la sezione 90.2.2.

```
1510001 .extern _main
1510002 .global ___mkargv
1510003 ;------------------------------------------------------------------------
1510004 ; Please note that, at the beginning, all the segment registers are
1510005 ; the same: CS==DS==ES==SS. But the data segments (DS, ES, SS) are meant
1510006 ; to be separated from the code, and they starts at: CS + __segoff.
1510007 ; The label "__segoff" is replaced by the linker with a constant value
1510008 ; (inside the code) with the segment offset to add to CS; this way
1510009 ; it is possibile to find where DS and ES should start.
1510010 ;------------------------------------------------------------------------
1510011 ; The following statement says that the code will start at "startup"
1510012 ; label.
1510013 ;------------------------------------------------------------------------
1510014 entry startup
1510015 ;------------------------------------------------------------------------
1510016 .text
1510017 ;------------------------------------------------------------------------
1510018 startup:
1510019     ;
1510020     ; Jump after initial data.
1510021     ;
1510022     jmp startup_code
1510023     ;
1510024 filler:
1510025     ;
1510026     ; After four bytes, from the start, there is the
1510027     ; magic number and other data.
1510028     ;
1510029     .space (0x0004 - (filler - startup))
1510030 magic:
1510031     ;
1510032     ; Add to "/etc/magic" the following line:
1510033     ;
1510034     ; 4  quad  0x6B65726E6F733136  os16 kernel
1510035     ;
1510036     .data4 0x6F733136   ; os16
1510037     .data4 0x6B65726E   ; kern
1510038     ;
1510039 segoff:                 ;
```

```
1510040      .data2 __segoff      ; These values, for a kernel image,
1510041  etext:                   ; are not used.
1510042      .data2 __etext       ;
1510043  edata:                   ;
1510044      .data2 __edata       ;
1510045  ebss:                    ;
1510046      .data2 __end         ;
1510047  stack_size:              ;
1510048      .data2 0x0000        ;
1510049      ;
1510050      ; At the next label, the work begins.
1510051      ;
1510052  .align 2
1510053  startup_code:
1510054      ;
1510055      ; Check where we are. If we are at segment 0x1000,
1510056      ; then move to 0x3000.
1510057      ;
1510058      mov cx, cs
1510059      xor cx, #0x1000
1510060      jcxz move_code_from_0x1000_to_0x3000
1510061      ;
1510062      ; Check where we are. If we are at segment 0x3000,
1510063      ; then move to 0x0050, preserving the IVT and the BDA.
1510064      ;
1510065      mov cx, cs
1510066      xor cx, #0x3000
1510067      jcxz move_code_from_0x3000_to_0x0050
1510068      ;
1510069      ; Check where we are. If we are at segment 0x1050,
1510070      ; then jump to the main code.
1510071      ;
1510072      mov cx, cs
1510073      xor cx, #0x1050
1510074      jcxz main_code
1510075      ;
1510076      ; Otherwise, just halt.
1510077      ;
1510078      hlt
1510079      jmp startup_code
1510080      ;
1510081  move_code_from_0x1000_to_0x3000:
1510082      ;
1510083      cld                  ; Clear direction flag.
1510084      mov   ax, #0x3000    ; Set ES as the destination segment.
1510085      mov   es, ax         ;
1510086      mov   ax, #0x1000    ; Set DS as the source segment.
1510087      mov   ds, ax         ;
```

```
1510088      ;
1510089      mov   cx, #0x8000   ; Move 32768 words = 65536 byte (64 Kibyte).
1510090      mov   si, #0x0000   ; DS:SI == Source pointer
1510091      mov   di, #0x0000   ; ES:DI == Destination pointer
1510092      rep
1510093      movsw               ; Copy the array of words
1510094      ;
1510095      mov   ax, #0x4000   ; Set ES as the destination segment.
1510096      mov   es, ax        ;
1510097      mov   ax, #0x2000   ; Set DS as the source segment.
1510098      mov   ds, ax        ;
1510099      ;
1510100      mov   cx, #0x8000   ; Move 32768 words = 65536 byte (64 Kibyte).
1510101      mov   si, #0x0000   ; DS:SI == Source pointer
1510102      mov   di, #0x0000   ; ES:DI == Destination pointer
1510103      rep
1510104      movsw               ; Copy the array of words
1510105      ;
1510106      jmp far #0x3000:#0x0000     ; Go to the new kernel copy.
1510107      ;
1510108  move_code_from_0x3000_to_0x0050:
1510109      cld                 ; Clear direction flag.
1510110      ;
1510111      ; Text (instructions) is moved at segment 0x1050 (address 0x10500).
1510112      ;
1510113      mov   ax, #0x1050   ; Set ES as the destination segment.
1510114      mov   es, ax        ;
1510115      mov   ax, #0x3000   ; Set DS as the source segment.
1510116      mov   ds, ax        ;
1510117      ;
1510118      mov   cx, #0x8000   ; Move 32768 words = 65536 byte (64 Kibyte).
1510119      mov   si, #0x0000   ; DS:SI == Source pointer
1510120      mov   di, #0x0000   ; ES:DI == Destination pointer
1510121      rep
1510122      movsw               ; Copy the array of words
1510123      ;
1510124      ; Data is moved at segment 0x0050 (address 0x00500), before the
1510125      ; text segment.
1510126      ;
1510127      mov   ax, #0x0050   ; Set ES as the destination segment.
1510128      mov   es, ax        ;
1510129      mov   ax, #0x3000   ; Calculate where is the data segment:
1510130      add   ax, #__segoff ; it is at 0x3000 + __segoff.
1510131      mov   ds, ax        ; Set DS as the source segment.
1510132      ;
1510133      mov   cx, #0x8000   ; Move 32768 words = 65536 byte (64 Kibyte).
1510134      mov   si, #0x0000   ; DS:SI == Source pointer
1510135      mov   di, #0x0000   ; ES:DI == Destination pointer
```

```
1510136     rep
1510137     movsw                ; Copy the array of words
1510138     ;
1510139     jmp far #0x1050:#0x0000 ; Go to the new kernel copy.
1510140     ;
1510141 ;-----------------------------------------------------------------------
1510142 main_code:
1510143     ;
1510144     ; Fix data segments!
1510145     ;
1510146     mov  ax, #0x0050
1510147     mov  ds, ax
1510148     mov  ss, ax
1510149     mov  es, ax
1510150     ;
1510151     ; Fix SP at the kernel stack bottom: the effective stack pointer
1510152     ; value should be 0x10000, but only 0x0000 can be written. At the
1510153     ; first push SP reaches 0xFFFE.
1510154     ;
1510155     mov sp, #0x0000
1510156     ;
1510157     ; Reset flags.
1510158     ;
1510159     push #0
1510160     popf
1510161     cli
1510162     ;
1510163     ; Call C main function, after kernel relocation and segments set up.
1510164     ;
1510165     push #0     ; This zero means NULL (envp[][] == NULL)
1510166     push #0     ; This zero means NULL (argv[][] == NULL)
1510167     push #0     ; This other zero means no arguments.
1510168     call _main
1510169     add  sp, #2
1510170     add  sp, #2
1510171     add  sp, #2
1510172     ;
1510173 .align 2
1510174 halt:
1510175     ;
1510176     ; It will never come back from the _kmain() call, but just for extra
1510177     ; security, loop forever.
1510178     ;
1510179     hlt
1510180     jmp halt
1510181     ;
1510182 ;-----------------------------------------------------------------------
1510183 .align 2
```

```
1510184 ___mkargv:      ; Symbol '___mkargv' is used by Bcc inside the function
1510185     ret         ; 'main()' and must be present for a successful
1510186                 ; compilation.
1510187 ;----------------------------------------------------------------------
1510188 .align 2
1510189 .data
1510190 ;
1510191 ;----------------------------------------------------------------------
1510192 .align 2
1510193 .bss
```

100.7.3 kernel/main/main.c

Si veda la sezione 99.6.

```
1520001 #include <kernel/main.h>
1520002 #include <errno.h>
1520003 #include <fcntl.h>
1520004 #include <kernel/main/build.h>
1520005 #include <kernel/diag.h>
1520006 #include <kernel/fs.h>
1520007 #include <kernel/ibm_i86.h>
1520008 #include <kernel/k_libc.h>
1520009 #include <kernel/proc.h>
1520010 #include <libgen.h>
1520011 #include <stdlib.h>
1520012 #include <sys/os16.h>
1520013 #include <sys/stat.h>
1520014 #include <sys/types.h>
1520015 #include <unistd.h>
1520016 //----------------------------------------------------------------------
1520017 int
1520018 main (int argc, char *argv[], char *envp[])
1520019 {
1520020     unsigned int key;
1520021     pid_t        pid;
1520022     char        *exec_argv[2];
1520023     int          status;
1520024     int          exit;
1520025     //
1520026     // Reset video and select the initial console.
1520027     //
1520028     tty_init ();
1520029     //
1520030     // Show compilation date and time.
1520031     //
1520032     k_printf ("os16 build %s ram %i Kibyte\n", BUILD_DATE, int12 ());
1520033     //
```

```
1520034      // Set up disk management.
1520035      //
1520036      dsk_setup ();
1520037      //
1520038      // Clear heap for diagnosis.
1520039      //
1520040      heap_clear ();
1520041      //
1520042      // Set up process management. Process set up need the file system
1520043      // root directory already available.
1520044      //
1520045      proc_init ();
1520046      //
1520047      // The kernel will run interactively.
1520048      //
1520049      menu ();
1520050      //
1520051      for (exit = 0; exit == 0;)
1520052        {
1520053          //
1520054          // While in kernel code, timer interrupt don't start the
1520055          // scheduler. The kernel must leave control to the scheduler
1520056          // via a null system call.
1520057          //
1520058          sys (SYS_0, NULL, 0);
1520059          //
1520060          // Back to work: read the keyboard from the TTY device.
1520061          //
1520062          dev_io ((pid_t) 0, DEV_TTY, DEV_READ, 0L, &key, 1, NULL);
1520063          //
1520064          // Depending on the key, do something.
1520065          //
1520066          if (key == 0)
1520067            {
1520068              //
1520069              // No key is ready in the buffer keyboard.
1520070              //
1520071              continue;
1520072            }
1520073          else
1520074            {
1520075              //
1520076              // Move back the cursor, so that next print will overwrite
1520077              // it.
1520078              //
1520079              k_printf ("\b");
1520080            }
1520081          //
```

```
1520082            // A key was pressed: start to check what it was.
1520083            //
1520084            switch (key)
1520085              {
1520086                case 'h':
1520087                    menu ();
1520088                    break;
1520089                case '1':
1520090                    k_kill ((pid_t) 1, SIGKILL);                   // init
1520091                    break;
1520092                case '2':
1520093                case '3':
1520094                case '4':
1520095                case '5':
1520096                case '6':
1520097                case '7':
1520098                case '8':
1520099                case '9':
1520100                    k_kill ((pid_t) (key - '0'),  SIGTERM);        // others
1520101                    break;
1520102                case 'A':
1520103                case 'B':
1520104                case 'C':
1520105                case 'D':
1520106                case 'E':
1520107                case 'F':
1520108                    k_kill ((pid_t) (key - 'A' + 10),  SIGTERM);   // others
1520109                    break;
1520110                case 'a':
1520111                    run ("/bin/aaa", NULL, NULL);
1520112                    break;
1520113                case 'b':
1520114                    run ("/bin/bbb", NULL, NULL);
1520115                    break;
1520116                case 'c':
1520117                    run ("/bin/ccc", NULL, NULL);
1520118                    break;
1520119                case 'f':
1520120                    print_file_list ();
1520121                    break;
1520122                case 'm':
1520123                    status = path_mount ((uid_t) 0, "/dev/dsk1", "/usr",
1520124                                         MOUNT_DEFAULT);
1520125                    if (status < 0)
1520126                      {
1520127                        k_perror (NULL);
1520128                      }
1520129                    break;
```

```
1520130                case 'M':
1520131                    status = path_umount ((uid_t) 0, "/usr");
1520132                    if (status < 0)
1520133                      {
1520134                        k_perror (NULL);
1520135                      }
1520136                    break;
1520137                case 'n':
1520138                    print_inode_list ();
1520139                    break;
1520140                case 'N':
1520141                    print_inode_zones_list ();
1520142                    break;
1520143                case 'l':
1520144                    k_kill ((pid_t) 1, SIGCHLD);
1520145                    break;
1520146                case 'p':
1520147                    k_printf ("\n");
1520148                    print_proc_list ();
1520149                    print_segments ();
1520150                    k_printf (" ");
1520151                    print_kmem ();
1520152                    k_printf (" ");
1520153                    print_time ();
1520154                    k_printf ("\n");
1520155                    print_mb_map ();
1520156                    k_printf ("\n");
1520157                    break;
1520158                case 'x':
1520159                    exit = 1;
1520160                    break;
1520161                case 'q':
1520162                    k_printf ("System halted!\n");
1520163                    return (0);
1520164                    break;
1520165              }
1520166          }
1520167        //
1520168        // Load init.
1520169        //
1520170        exec_argv[0] = "/bin/init";
1520171        exec_argv[1] = NULL;
1520172        pid = run ("/bin/init", exec_argv, NULL);
1520173        //
1520174        // Just sleep.
1520175        //
1520176        while (1)
1520177          {
```

```
1520178             sys (SYS_0, NULL, 0);
1520179           }
1520180         //
1520181         k_printf ("System halted!\n");
1520182         return (0);
1520183     }
```

100.7.4 kernel/main/menu.c

Si veda la sezione 99.6.

```
1530001     #include <kernel/main.h>
1530002     #include <kernel/k_libc.h>
1530003     //----------------------------------------------------------------------
1530004     void
1530005     menu (void)
1530006     {
1530007         k_printf (
1530008     ".------------------------------------------------------------------.\n"
1530009     "| [h]      show this menu                                          |\n"
1530010     "| [p]      process status and memory map                           |\n"
1530011     "| [1]..[9] kill process  1 to 9                                    |\n"
1530012     "| [A]..[F] kill process 10 to 15                                   |\n"
1530013     "| [l]      send SIGCHLD to process 1                               |\n"
1530014     "| [a]..[c] run programs '/bin/aaa' to '/bin/ccc' in parallel       |\n"
1530015     "| [f]      system file status                                      |\n"
1530016     "| [n], [N] list of active inodes                                   |\n"
1530017     "| [m], [M] mount/umount '/dev/dsk1' at '/usr/'                     |\n"
1530018     "| [x]      exit interaction with kernel and start '/bin/init'      |\n"
1530019     "| [q]      quit kernel                                             |\n"
1530020     "`------------------------------------------------------------------'\n"
1530021     );
1530022
1530023     }
```

100.7.5 kernel/main/run.c

Si veda la sezione 99.6.

```
1540001     #include <kernel/main.h>
1540002     #include <kernel/proc.h>
1540003     #include <kernel/k_libc.h>
1540004     #include <unistd.h>
1540005     //----------------------------------------------------------------------
1540006     pid_t
1540007     run (char *path, char *argv[], char *envp[])
1540008     {
```

```
1540009     pid_t pid;
1540010     //
1540011     pid = fork ();
1540012     if (pid == -1)
1540013       {
1540014         k_perror (NULL);
1540015       }
1540016     else if (pid == 0)
1540017       {
1540018         execve (path, argv, envp);
1540019         k_perror (NULL);
1540020         _exit (0);
1540021       }
1540022     return (pid);
1540023 }
1540024
```

100.8 os16: «kernel/memory.h»

Si veda la sezione 99.7.

```
1550001 #ifndef _KERNEL_MEMORY_H
1550002 #define _KERNEL_MEMORY_H        1
1550003
1550004 #include <stdint.h>
1550005 #include <stddef.h>
1550006 #include <sys/types.h>
1550007 //----------------------------------------------------------------------
1550008 #define MEM_BLOCK_SIZE      256   // 0x0100
1550009 #define MEM_MAX_BLOCKS     2560   // 655360/256 = 0xA0000/0x0100 = 0x0A00
1550010
1550011 extern uint16_t mb_table[MEM_MAX_BLOCKS/16]; // Memory blocks map.
1550012 //----------------------------------------------------------------------
1550013 typedef unsigned long int addr_t;
1550014 typedef unsigned int      segment_t;
1550015 typedef unsigned int      offset_t;
1550016 //----------------------------------------------------------------------
1550017 typedef struct {
1550018     addr_t    address;
1550019     segment_t segment;
1550020     size_t    size;
1550021 } memory_t;
1550022 //----------------------------------------------------------------------
1550023 addr_t    address        (segment_t segment, offset_t offset);
1550024 //----------------------------------------------------------------------
1550025 uint16_t *mb_reference   (void);
1550026 ssize_t   mb_alloc       (addr_t address, size_t size);
1550027 void      mb_free        (addr_t address, size_t size);
```

```
1550028 int        mb_alloc_size (size_t size, memory_t *allocated);
1550029 //----------------------------------------------------------------------
1550030 void       mem_copy      (addr_t orig, addr_t dest, size_t size);
1550031 size_t     mem_read      (addr_t start, void *buffer, size_t size);
1550032 size_t     mem_write     (addr_t start, void *buffer, size_t size);
1550033 //----------------------------------------------------------------------
1550034 
1550035 #endif
```

100.8.1 kernel/memory/address.c

Si veda la sezione 99.7.

```
1560001 #include <kernel/memory.h>
1560002 //----------------------------------------------------------------------
1560003 addr_t
1560004 address (segment_t segment, offset_t offset)
1560005 {
1560006     addr_t a;
1560007     a  = segment;
1560008     a *= 16;
1560009     a += offset;
1560010     return (a);
1560011 }
1560012 
1560013 
1560014 
1560015 
1560016 
1560017 
1560018 
1560019 
1560020 
```

100.8.2 kernel/memory/mb_alloc.c

Si veda la sezione 99.7.

```
1570001 #include <kernel/memory.h>
1570002 #include <kernel/ibm_i86.h>
1570003 #include <sys/os16.h>
1570004 #include <kernel/k_libc.h>
1570005 //----------------------------------------------------------------------
1570006 static int mb_block_set1   (int block);
1570007 //----------------------------------------------------------------------
1570008 ssize_t
1570009 mb_alloc (addr_t address, size_t size)
```

```
1570010 {
1570011     unsigned int bstart;
1570012     unsigned int bsize;
1570013     unsigned int bend;
1570014     unsigned int i;
1570015     ssize_t      allocated = 0;
1570016     addr_t       block_address;
1570017 
1570018     if (size == 0)
1570019       {
1570020         //
1570021         // Zero means the maximum size.
1570022         //
1570023         bsize  = 0x10000L / MEM_BLOCK_SIZE;
1570024       }
1570025     else
1570026       {
1570027         bsize  = size / MEM_BLOCK_SIZE;
1570028       }
1570029 
1570030     bstart = address / MEM_BLOCK_SIZE;
1570031 
1570032     if (size % MEM_BLOCK_SIZE)
1570033       {
1570034         bend = bstart + bsize;
1570035       }
1570036     else
1570037       {
1570038         bend = bstart + bsize - 1;
1570039       }
1570040 
1570041     for (i = bstart; i <= bend; i++)
1570042       {
1570043         if (mb_block_set1 (i))
1570044           {
1570045             allocated += MEM_BLOCK_SIZE;
1570046           }
1570047         else
1570048           {
1570049             block_address = i;
1570050             block_address *= MEM_BLOCK_SIZE;
1570051             k_printf ("Kernel alert: mem block %04x, at address ", i);
1570052             k_printf ("%05lx, already allocated!\n", block_address);
1570053             break;
1570054           }
1570055       }
1570056     return (allocated);
1570057 }
```

```
1570058
1570059 //----------------------------------------------------------------------
1570060 static int
1570061 mb_block_set1 (int block)
1570062 {
1570063     int i          = block / 16;
1570064     int j          = block % 16;
1570065     uint16_t mask = 0x8000 >> j;
1570066     if (mb_table[i] & mask)
1570067       {
1570068         return (0);     // The block is already set to 1 inside the map!
1570069       }
1570070     else
1570071       {
1570072         mb_table[i] = mb_table[i] | mask;
1570073         return (1);
1570074       }
1570075 }
```

100.8.3 kernel/memory/mb_alloc_size.c

Si veda la sezione 99.7.

```
1580001 #include <kernel/memory.h>
1580002 #include <kernel/ibm_i86.h>
1580003 #include <sys/os16.h>
1580004 #include <errno.h>
1580005 //----------------------------------------------------------------------
1580006 static int mb_block_status (int block);
1580007 //----------------------------------------------------------------------
1580008 int
1580009 mb_alloc_size (size_t size, memory_t *allocated)
1580010 {
1580011     unsigned int bsize;
1580012     unsigned int i;
1580013     unsigned int j;
1580014     unsigned int found = 0;
1580015     addr_t       alloc_addr;
1580016     ssize_t      alloc_size;
1580017
1580018     if (size == 0)
1580019       {
1580020         //
1580021         // Zero means the maximum size.
1580022         //
1580023         bsize = 0x10000L / MEM_BLOCK_SIZE;
1580024       }
1580025     else if (size % MEM_BLOCK_SIZE)
```

```
1580026         {
1580027           bsize = size / MEM_BLOCK_SIZE + 1;
1580028         }
1580029       else
1580030         {
1580031           bsize = size / MEM_BLOCK_SIZE;
1580032         }
1580033 
1580034       for (i = 0; i < (MEM_MAX_BLOCKS - bsize) && !found; i++)
1580035         {
1580036           for (j = 0; j < bsize; j++)
1580037             {
1580038               found = !mb_block_status (i+j);
1580039               if (!found)
1580040                 {
1580041                   i += j;
1580042                   break;
1580043                 }
1580044             }
1580045         }
1580046 
1580047       if (found && (j == bsize))
1580048         {
1580049           alloc_addr  = i - 1;
1580050           alloc_addr *= MEM_BLOCK_SIZE;
1580051           alloc_size  = bsize * MEM_BLOCK_SIZE;
1580052           alloc_size  = mb_alloc (alloc_addr, (size_t) alloc_size);
1580053           if (alloc_size <= 0)
1580054             {
1580055               errset (ENOMEM);
1580056               return (-1);
1580057             }
1580058           else if (alloc_size < size)
1580059             {
1580060               mb_free (alloc_addr, (size_t) alloc_size);
1580061               errset (ENOMEM);
1580062               return (-1);
1580063             }
1580064           else
1580065             {
1580066               allocated->address = alloc_addr;
1580067               allocated->segment = alloc_addr / 16;
1580068               allocated->size    = (size_t) alloc_size;
1580069             }
1580070           return (0);
1580071         }
1580072       else
1580073        {
```

```
1580074             errset (ENOMEM);
1580075             return (-1);
1580076           }
1580077   }
1580078   //----------------------------------------------------------------------
1580079   static int
1580080   mb_block_status (int block)
1580081   {
1580082       int i          = block / 16;
1580083       int j          = block % 16;
1580084       uint16_t mask = 0x8000 >> j;
1580085       return ((int) (mb_table[i] & mask));
1580086   }
```

100.8.4 kernel/memory/mb_free.c

Si veda la sezione 99.7.

```
1590001   #include <kernel/memory.h>
1590002   #include <kernel/ibm_i86.h>
1590003   #include <sys/os16.h>
1590004   #include <kernel/k_libc.h>
1590005   //----------------------------------------------------------------------
1590006   static int mb_block_set0   (int block);
1590007   //----------------------------------------------------------------------
1590008   void
1590009   mb_free (addr_t address, size_t size)
1590010   {
1590011       unsigned int bstart;
1590012       unsigned int bsize;
1590013       unsigned int bend;
1590014       unsigned int i;
1590015       addr_t       block_address;
1590016       if (size == 0)
1590017         {
1590018           //
1590019           // Zero means the maximum size.
1590020           //
1590021           bsize  = 0x10000L / MEM_BLOCK_SIZE;
1590022         }
1590023       else
1590024         {
1590025           bsize  = size / MEM_BLOCK_SIZE;
1590026         }
1590027
1590028       bstart = address / MEM_BLOCK_SIZE;
1590029
1590030       if (size % MEM_BLOCK_SIZE)
```

```
1590031         {
1590032           bend = bstart + bsize;
1590033         }
1590034       else
1590035         {
1590036           bend = bstart + bsize - 1;
1590037         }
1590038 
1590039       for (i = bstart; i <= bend; i++)
1590040         {
1590041           if (mb_block_set0 (i))
1590042             {
1590043               ;
1590044             }
1590045           else
1590046             {
1590047               block_address = i;
1590048               block_address *= MEM_BLOCK_SIZE;
1590049               k_printf ("Kernel alert: mem block %04x, at address ", i);
1590050               k_printf ("%05lx, already released!\n", block_address);
1590051             }
1590052         }
1590053 }
1590054 //----------------------------------------------------------------------
1590055 static int
1590056 mb_block_set0 (int block)
1590057 {
1590058     int i          = block / 16;
1590059     int j          = block % 16;
1590060     uint16_t mask = 0x8000 >> j;
1590061     if (mb_table[i] & mask)
1590062       {
1590063         mb_table[i] = mb_table[i] & ~mask;
1590064         return (1);
1590065       }
1590066     else
1590067       {
1590068         return (0);     // The block is already set to 0 inside the map!
1590069       }
1590070 }
```

100.8.5 kernel/memory/mb_reference.c

«

Si veda la sezione 99.7.

```
1600001 #include <stdint.h>
1600002 #include <kernel/memory.h>
1600003 //----------------------------------------------------------------------
1600004 uint16_t *
1600005 mb_reference (void)
1600006 {
1600007     return mb_table;
1600008 }
1600009
```

100.8.6 kernel/memory/mb_table.c

«

Si veda la sezione 99.7.

```
1610001 #include <kernel/memory.h>
1610002 #include <stdint.h>
1610003 //----------------------------------------------------------------------
1610004 uint16_t mb_table[MEM_MAX_BLOCKS/16]; // Memory blocks map.
1610005 //----------------------------------------------------------------------
```

100.8.7 kernel/memory/mem_copy.c

«

Si veda la sezione 99.7.

```
1620001 #include <kernel/memory.h>
1620002 #include <kernel/ibm_i86.h>
1620003 #include <sys/os16.h>
1620004 //----------------------------------------------------------------------
1620005 void
1620006 mem_copy (addr_t orig, addr_t dest, size_t size)
1620007 {
1620008     segment_t seg_orig = orig / 16;
1620009     offset_t  off_orig = orig % 16;
1620010     segment_t seg_dest = dest / 16;
1620011     offset_t  off_dest = dest % 16;
1620012     ram_copy (seg_orig, off_orig, seg_dest, off_dest, size);
1620013 }
```

100.8.8 kernel/memory/mem_read.c

«

Si veda la sezione 99.7.

```
1630001 #include <kernel/memory.h>
1630002 #include <kernel/ibm_i86.h>
1630003 #include <sys/os16.h>
1630004 //----------------------------------------------------------------------
1630005 size_t
1630006 mem_read (addr_t start, void *buffer, size_t size)
1630007 {
1630008     unsigned int       segment = start / 16;
1630009     unsigned int       offset  = start % 16;
1630010     unsigned long int end;
1630011     end  = start;
1630012     end += size;
1630013     if (end > 0x000FFFFFL)
1630014       {
1630015         size = 0x000FFFFFL - start;
1630016       }
1630017     ram_copy (segment, offset, seg_d (), (unsigned int) buffer, size);
1630018     return (size);
1630019 }
```

100.8.9 kernel/memory/mem_write.c

«

Si veda la sezione 99.7.

```
1640001 #include <kernel/memory.h>
1640002 #include <kernel/ibm_i86.h>
1640003 #include <sys/os16.h>
1640004 //----------------------------------------------------------------------
1640005 size_t
1640006 mem_write (addr_t start, void *buffer, size_t size)
1640007 {
1640008     unsigned int       segment = start / 16;
1640009     unsigned int       offset  = start % 16;
1640010     unsigned long int end;
1640011     end  = start;
1640012     end += size;
1640013     if (end > 0x000FFFFFL)
1640014       {
1640015         size = 0x000FFFFFL - start;
1640016       }
1640017     ram_copy (seg_d (), (unsigned int) buffer, segment, offset, size);
1640018     return (size);
1640019 }
```

100.9 os16: «kernel/proc.h»

Si veda la sezione 99.8.

```
1650001 #ifndef _KERNEL_PROC_H
1650002 #define _KERNEL_PROC_H  1
1650003
1650004 #include <kernel/devices.h>
1650005 #include <kernel/memory.h>
1650006 #include <kernel/fs.h>
1650007 #include <kernel/tty.h>
1650008 #include <sys/types.h>
1650009 #include <sys/stat.h>
1650010 #include <sys/os16.h>
1650011 #include <stddef.h>
1650012 #include <stdint.h>
1650013 #include <time.h>
1650014
1650015 //----------------------------------------------------------------------
1650016 #define CLOCK_FREQUENCY_DIVISOR     65535 // [1]
1650017 //
1650018 // [1]
1650019 // Internal clock frequency is (3579545/3) Hz.
1650020 // This value is divided by 65535 (0xFFFF) giving 18.2 Hz.
1650021 // The divisor value, 65535, if fixed!
1650022 //
1650023 //----------------------------------------------------------------------
1650024 #define PROC_EMPTY                      0
1650025 #define PROC_CREATED                    1
1650026 #define PROC_READY                      2
1650027 #define PROC_RUNNING                    3
1650028 #define PROC_SLEEPING                   4
1650029 #define PROC_ZOMBIE                     5
1650030 //----------------------------------------------------------------------
1650031 #define MAGIC_OS16              0x6F733136L  // os16
1650032 #define MAGIC_OS16_APPL         0x6170706CL  // appl
1650033 #define MAGIC_OS16_KERN         0x6B65726EL  // kern
1650034 //----------------------------------------------------------------------
1650035 #define PROCESS_MAX  16                 // Process slots.
1650036
1650037 typedef struct {
1650038     pid_t               ppid;           // Parent PID.
1650039     pid_t               pgrp;           // Process group ID.
1650040     uid_t               uid;            // Real user ID
1650041     uid_t               euid;           // Effective user ID.
1650042     uid_t               suid;           // Saved user ID.
1650043     dev_t               device_tty;     // Controlling terminal.
1650044     char                path_cwd[PATH_MAX];
1650045                                         // Working directory path.
```

```
1650046     inode_t          *inode_cwd;        // Working directory inode.
1650047     int               umask;            // File creation mask.
1650048     unsigned long int sig_status;       // Active signals.
1650049     unsigned long int sig_ignore;       // Signals to be ignored.
1650050     clock_t           usage;            // Clock ticks CPU time usage.
1650051     unsigned int      status;
1650052     int               wakeup_events;    // Wake up for something.
1650053     int               wakeup_signal;    // Signal waited.
1650054     unsigned int      wakeup_timer;     // Seconds to wait for.
1650055     addr_t            address_i;
1650056     segment_t         segment_i;
1650057     size_t            size_i;
1650058     addr_t            address_d;
1650059     segment_t         segment_d;
1650060     size_t            size_d;
1650061     uint16_t          sp;
1650062     int               ret;
1650063     char              name[PATH_MAX];
1650064     fd_t              fd[FOPEN_MAX];
1650065 } proc_t;
1650066
1650067 extern proc_t   proc_table[PROCESS_MAX];
1650068 //----------------------------------------------------------------------
1650069 typedef struct {
1650070     uint32_t filler0;
1650071     uint32_t magic0;
1650072     uint32_t magic1;
1650073     uint16_t segoff;
1650074     uint16_t etext;
1650075     uint16_t edata;
1650076     uint16_t ebss;
1650077     uint16_t ssize;
1650078 } header_t;
1650079 //----------------------------------------------------------------------
1650080 void         _ivt_load          (void);
1650081 #define      ivt_load()         (_ivt_load ())
1650082 void         proc_init          (void);
1650083 void         proc_scheduler     (uint16_t *sp, segment_t *segment);
1650084 void         sysroutine         (uint16_t *sp, segment_t *segment,
1650085                                  uint16_t syscallnr, uint16_t msg_off,
1650086                                  uint16_t msg_size);
1650087 proc_t      *proc_reference     (pid_t pid);
1650088 //----------------------------------------------------------------------
1650089 int          proc_sys_exec      (uint16_t *sp, segment_t *segment_d,
1650090                                  pid_t pid, const char *path,
1650091                                  unsigned int argc, char *arg_data,
1650092                                  unsigned int envc, char *env_data);
1650093 void         proc_sys_exit      (pid_t pid, int status);
```

```
1650094 pid_t        proc_sys_fork      (pid_t ppid, uint16_t sp);
1650095 int          proc_sys_kill      (pid_t pid_killer, pid_t pid_target,
1650096                                  int sig);
1650097 int          proc_sys_seteuid   (pid_t pid, uid_t euid);
1650098 int          proc_sys_setuid    (pid_t pid, uid_t uid);
1650099 sighandler_t proc_sys_signal    (pid_t pid, int sig,
1650100                                  sighandler_t handler);
1650101 pid_t        proc_sys_wait      (pid_t pid, int *status);
1650102 //----------------------------------------------------------------------
1650103 void         proc_dump_memory   (pid_t pid, addr_t address,
1650104                                  size_t size, char *name);
1650105 void         proc_available     (pid_t pid);
1650106 pid_t        proc_find          (segment_t segment_d);
1650107 void         proc_sch_signals   (void);
1650108 void         proc_sch_terminals (void);
1650109 void         proc_sch_timers    (void);
1650110 void         proc_sig_chld      (pid_t parent, int sig);
1650111 void         proc_sig_cont      (pid_t pid, int sig);
1650112 void         proc_sig_core      (pid_t pid, int sig);
1650113 int          proc_sig_ignore    (pid_t pid, int sig);
1650114 void         proc_sig_off       (pid_t pid, int sig);
1650115 void         proc_sig_on        (pid_t pid, int sig);
1650116 int          proc_sig_status    (pid_t pid, int sig);
1650117 void         proc_sig_stop      (pid_t pid, int sig);
1650118 void         proc_sig_term      (pid_t pid, int sig);
1650119
1650120 #endif
```

100.9.1 kernel/proc/_isr.s

«

Si veda la sezione 99.8.1.

```
1660001 .extern _proc_scheduler
1660002 .extern _sysroutine
1660003 .global __ksp
1660004 .global __clock_ticks
1660005 .global __clock_seconds
1660006 .global isr_1C
1660007 .global isr_80
1660008 ;-----------------------------------------------------------------------
1660009 ; The kernel code segment starts at 0x10500 (segment 0x1050).
1660010 ; The kernel data segments start at 0x00500 (segment 0x0050).
1660011 ; To switch to the kernel data segments, DS, ES and SS are set to
1660012 ; 0x0050. To identify the kernel context, the DS register is checked:
1660013 ; if it is equal to 0x0050, it is the kernel.
1660014 ;-----------------------------------------------------------------------
1660015 .data
1660016 ;-----------------------------------------------------------------------
```

```
1660017 .align 2
1660018 proc_ss_0:       .word 0x0000
1660019 proc_sp_0:       .word 0x0000
1660020 proc_ss_1:       .word 0x0000
1660021 proc_sp_1:       .word 0x0000
1660022 proc_syscallnr:  .word 0x0000
1660023 proc_msg_offset: .word 0x0000
1660024 proc_msg_size:   .word 0x0000
1660025 __ksp:           .word 0x0000
1660026 __clock_ticks:
1660027 ticks_lo:        .word 0x0000
1660028 ticks_hi:        .word 0x0000
1660029 __clock_seconds:
1660030 seconds_lo:      .word 0x0000
1660031 seconds_hi:      .word 0x0000
1660032 ;-----------------------------------------------------------------------
1660033 .text
1660034 ;-----------------------------------------------------------------------
1660035 ; IRQ 0: "timer".
1660036 ; IRQ 0 is associated to INT 8, and after the BIOS work is done,
1660037 ; INT 1C is called. Standard INT 1C has nothing to do, but is
1660038 ; useful to call extra work for the timer. As the original BIOS
1660039 ; interrupts are used, the INT 1C is reprogrammed, keeping intact
1660040 ; the Standard INT 8.
1660041 ;-----------------------------------------------------------------------
1660042 .align 2
1660043 isr_1C:
1660044     ;-------------------------------------------------------------------
1660045     ; Inside the process stack, the CPU already put:
1660046     ;
1660047     ;       [omissis]
1660048     ;       push flags
1660049     ;       push cs
1660050     ;       push ip
1660051     ;-------------------------------------------------------------------
1660052     ;
1660053     ; Save into process stack:
1660054     ;
1660055     push   es    ; extra segment
1660056     push   ds    ; data segment
1660057     push   di    ; destination index
1660058     push   si    ; source index
1660059     push   bp    ; base pointer
1660060     push   bx    ; BX
1660061     push   dx    ; DX
1660062     push   cx    ; CX
1660063     push   ax    ; AX
1660064     ;
```

```
1660065     ; Set the data segments to the kernel data area,
1660066     ; so that the following variables can be accessed.
1660067     ;
1660068     mov ax, #0x0050  ; DS and ES.
1660069     mov ds, ax       ;
1660070     mov es, ax       ;
1660071     ;
1660072     ; Increment time counters, to keep time.
1660073     ;
1660074     add ticks_lo, #1    ; Clock ticks counter.
1660075     adc ticks_hi, #0    ;
1660076     ;
1660077     mov dx, ticks_hi    ;
1660078     mov ax, ticks_lo    ; DX := ticks % 18
1660079     mov cx, #18         ;
1660080     div cx              ;
1660081     mov ax, #0          ; If the ticks value can be divided by 18,
1660082     cmp ax, dx          ; the seconds is incremented by 1.
1660083     jnz L1              ;
1660084     add seconds_lo, #1  ;
1660085     adc seconds_hi, #0  ;
1660086     ;
1660087 L1: ; Save process stack registers into kernel data segment.
1660088     ;
1660089     mov proc_ss_0, ss   ; Save process stack segment.
1660090     mov proc_sp_0, sp   ; Save process stack pointer.
1660091     ;
1660092     ; Check if it is already in kernel mode.
1660093     ;
1660094     mov dx, proc_ss_0
1660095     mov ax, #0x0050     ; Kernel data area.
1660096     cmp dx, ax
1660097     je L2
1660098     ;
1660099     ; If we are here, a user process was interrupted.
1660100     ; Switch to the kernel stack.
1660101     ;
1660102     mov ax, #0x0050     ; Kernel data area.
1660103     mov ss, ax
1660104     mov sp, __ksp
1660105     ;
1660106     ; Call the scheduler.
1660107     ;
1660108     push #proc_ss_0     ; &proc_ss_0
1660109     push #proc_sp_0     ; &proc_sp_0
1660110     call _proc_scheduler
1660111     add  sp, #2
1660112     add  sp, #2
```

```
1660113      ;
1660114      ; Restore process stack registers from kernel data segment.
1660115      ;
1660116      mov ss, proc_ss_0   ; Restore process stack segment.
1660117      mov sp, proc_sp_0   ; Restore process stack pointer.
1660118      ;
1660119  L2: ; Restore from process stack:
1660120      ;
1660121      pop    ax
1660122      pop    cx
1660123      pop    dx
1660124      pop    bx
1660125      pop    bp
1660126      pop    si
1660127      pop    di
1660128      pop    ds
1660129      pop    es
1660130      ;
1660131      ; Return from interrupt: will restore CS:IP and FLAGS
1660132      ; from process stack.
1660133      ;
1660134      iret
1660135  ;------------------------------------------------------------------------
1660136  ; Syscall.
1660137  ;------------------------------------------------------------------------
1660138  .align 2
1660139  isr_80:
1660140      ;-------------------------------------------------------------------
1660141      ; Inside the process stack, we already have:
1660142      ;       push #message_size
1660143      ;       push &message_structure     ; the relative address of it
1660144      ;       push #syscall_number
1660145      ;       push #back_address  ; made by a call to _sys function
1660146      ;       push flags          ; made by int #0x80
1660147      ;       push cs             ; made by int #0x80
1660148      ;       push ip             ; made by int #0x80
1660149      ;-------------------------------------------------------------------
1660150      ;
1660151      ; Save into process stack:
1660152      ;
1660153      push   es   ; extra segment
1660154      push   ds   ; data segment
1660155      push   di   ; destination index
1660156      push   si   ; source index
1660157      push   bp   ; base pointer
1660158      push   bx   ; BX
1660159      push   dx   ; DX
1660160      push   cx   ; CX
```

```
1660161       push    ax    ; AX
1660162       ;
1660163       ; Set the data segments to the kernel data area,
1660164       ; so that the following variables can be accessed.
1660165       ;
1660166       mov ax, #0x0050     ; DS and ES.
1660167       mov ds, ax          ;
1660168       mov es, ax          ;
1660169       ;
1660170       ; Save process stack registers into kernel data segment.
1660171       ;
1660172       mov proc_ss_1, ss   ; Save process stack segment.
1660173       mov proc_sp_1, sp   ; Save process stack pointer.
1660174       ;
1660175       ; Save some more data, from the system call.
1660176       ;
1660177       mov bp, sp
1660178       mov ax, +26[bp]
1660179       mov proc_syscallnr,  ax
1660180       mov ax, +28[bp]
1660181       mov proc_msg_offset, ax
1660182       mov ax, +30[bp]
1660183       mov proc_msg_size,   ax
1660184       ;
1660185       ; Check if it is already the kernel stack.
1660186       ;
1660187       mov dx, ss
1660188       mov ax, #0x0050     ; Kernel data area.
1660189       cmp dx, ax
1660190       jne L3
1660191       ;
1660192       ; It is already the kernel stack, so, the variable "_ksp" is
1660193       ; aligned to current stack pointer. This way, the first syscall
1660194       ; can work without having to set the "_ksp" variable to some
1660195       ; reasonable value.
1660196       ;
1660197       mov __ksp, sp
1660198       ;
1660199   L3: ; Switch to the kernel stack.
1660200       ;
1660201       mov ax, #0x0050     ; Kernel data area.
1660202       mov ss, ax
1660203       mov sp, __ksp
1660204       ;
1660205       ; Call the external hardware interrupt handler.
1660206       ;
1660207       push proc_msg_size
1660208       push proc_msg_offset
```

```
1660209     push proc_syscallnr
1660210     push #proc_ss_1      ; &proc_ss_1
1660211     push #proc_sp_1      ; &proc_sp_1
1660212     call _sysroutine
1660213     add  sp, #2
1660214     add  sp, #2
1660215     add  sp, #2
1660216     add  sp, #2
1660217     add  sp, #2
1660218     ;
1660219     ; Restore process stack registers from kernel data segment.
1660220     ;
1660221     mov ss, proc_ss_1    ; Restore process stack segment.
1660222     mov sp, proc_sp_1    ; Restore process stack pointer.
1660223     ;
1660224     ; Restore from process stack:
1660225     ;
1660226     pop    ax
1660227     pop    cx
1660228     pop    dx
1660229     pop    bx
1660230     pop    bp
1660231     pop    si
1660232     pop    di
1660233     pop    ds
1660234     pop    es
1660235     ;
1660236     ; Return from interrupt: will restore CS:IP and FLAGS
1660237     ; from process stack.
1660238     ;
1660239     iret
```

100.9.2 kernel/proc/_ivt_load.s

«

Si veda la sezione 99.8.2.

```
1670001 .extern isr_1C
1670002 .extern isr_80
1670003 .global __ivt_load
1670004 ;----------------------------------------------------------------------
1670005 .text
1670006 ;----------------------------------------------------------------------
1670007 ; Load IVT.
1670008 ;
1670009 ; Currently, only the timer function and the syscall are loaded.
1670010 ;----------------------------------------------------------------------
1670011 .align 2
1670012 __ivt_load:
```

```
1670013     enter #0, #0              ; No local variables.
1670014     pushf
1670015     cli
1670016     pusha
1670017     ;
1670018     mov   ax,   #0              ; Change the DS segment to 0.
1670019     mov   ds,   ax              ;
1670020     ;
1670021     mov   bx,   #112            ; Timer         INT 0x08 (8) --> 0x1C
1670022     mov   [bx], #isr_1C         ; offset
1670023     mov   bx,   #114            ;
1670024     mov   [bx], cs              ; segment
1670025     ;
1670026     mov   bx,   #512            ; Syscall       INT 0x80 (128)
1670027     mov   [bx], #isr_80         ; offset
1670028     mov   bx,   #514            ;
1670029     mov   [bx], cs              ; segment
1670030     ;
1670031     mov   ax,   #0x0050         ; Put the DS segment back to the right
1670032     mov   ds,   ax              ; value.
1670033     ;
1670034     ;
1670035     ;
1670036     popa
1670037     popf
1670038     leave
1670039     ret
```

100.9.3 kernel/proc/proc_available.c

Si veda la sezione 99.8.3.

```
1680001 #include <kernel/proc.h>
1680002 //----------------------------------------------------------------------
1680003 void
1680004 proc_available (pid_t pid)
1680005 {
1680006     proc_table[pid].ppid          = -1;
1680007     proc_table[pid].pgrp          = -1;
1680008     proc_table[pid].uid           = -1;
1680009     proc_table[pid].euid          = -1;
1680010     proc_table[pid].suid          = -1;
1680011     proc_table[pid].sig_status    = 0;
1680012     proc_table[pid].sig_ignore    = 0;
1680013     proc_table[pid].usage         = 0;
1680014     proc_table[pid].status        = PROC_EMPTY;
1680015     proc_table[pid].wakeup_events = 0;
1680016     proc_table[pid].wakeup_signal = 0;
```

```
1680017     proc_table[pid].wakeup_timer  = 0;
1680018     proc_table[pid].segment_i     = -1;
1680019     proc_table[pid].address_i     = -1L;
1680020     proc_table[pid].size_i        = -1;
1680021     proc_table[pid].segment_d     = -1;
1680022     proc_table[pid].address_d     = -1L;
1680023     proc_table[pid].size_d        = -1;
1680024     proc_table[pid].sp            = 0;
1680025     proc_table[pid].ret           = 0;
1680026     proc_table[pid].inode_cwd     = 0;
1680027     proc_table[pid].path_cwd[0]   = 0;
1680028     proc_table[pid].umask         = 0;
1680029     proc_table[pid].name[0]       = 0;
1680030 }
```

100.9.4 kernel/proc/proc_dump_memory.c

«

Si veda la sezione 99.8.4.

```
1690001 #include <kernel/proc.h>
1690002 #include <fcntl.h>
1690003 //----------------------------------------------------------------------
1690004 void
1690005 proc_dump_memory (pid_t pid, addr_t address, size_t size, char *name)
1690006 {
1690007     int      fdn;
1690008     char     buffer[SB_BLOCK_SIZE];
1690009     ssize_t size_written;
1690010     ssize_t size_written_total;
1690011     ssize_t size_read;
1690012     ssize_t size_read_total;
1690013     ssize_t size_total;
1690014     //
1690015     // Dump the code segment to disk.
1690016     //
1690017     fdn = fd_open (pid, name, (O_WRONLY|O_CREAT|O_TRUNC),
1690018                         (mode_t) (S_IFREG|00644));
1690019     if (fdn < 0)
1690020       {
1690021         //
1690022         // There is a problem: just let it go.
1690023         //
1690024         return;
1690025       }
1690026     //
1690027     // Fix size: (size_t) 0 is equivalent to (ssize_t) 0x10000.
1690028     //
1690029     size_total = size;
```

```
1690030     if (size_total == 0)
1690031       {
1690032         size_total = 0x10000;
1690033       }
1690034     //
1690035     // Read the memory and write it to disk.
1690036     //
1690037     for (size_read = 0, size_read_total = 0;
1690038          size_read_total < size_total;
1690039          size_read_total += size_read, address += size_read)
1690040       {
1690041         size_read = mem_read (address, buffer, SB_BLOCK_SIZE);
1690042         //
1690043         for (size_written = 0, size_written_total = 0;
1690044              size_written_total < size_read;
1690045              size_written_total += size_written)
1690046           {
1690047             size_written = fd_write (pid, fdn,
1690048                                &buffer[size_written_total],
1690049                                (size_t) (size_read - size_written_total));
1690050             //
1690051             if (size_written < 0)
1690052               {
1690053                 fd_close (pid, fdn);
1690054                 return;
1690055               }
1690056           }
1690057       }
1690058     fd_close (pid, fdn);
1690059 }
```

100.9.5 kernel/proc/proc_find.c

Si veda la sezione 99.8.5.

```
1700001 #include <kernel/proc.h>
1700002 #include <kernel/k_libc.h>
1700003 #include <kernel/diag.h>
1700004 //----------------------------------------------------------------------
1700005 pid_t
1700006 proc_find (segment_t segment_d)
1700007 {
1700008     int     pid;
1700009     addr_t address_d;
1700010     for (pid = 0; pid < PROCESS_MAX; pid++)
1700011       {
1700012         if (proc_table[pid].segment_d == segment_d)
1700013           {
```

```
1700014                 break;
1700015               }
1700016           }
1700017         if (pid >= PROCESS_MAX)
1700018           {
1700019             address_d  = segment_d;
1700020             address_d *= 16;
1700021             k_printf ("\n"
1700022                       "Kernel panic: cannot find the interrupted process "
1700023                       "inside the process table. "
1700024                       "The wanted process has data segment 0x%04x \n"
1700025                       "(effective address %05lx)!\n",
1700026                       (unsigned int) segment_d, address_d);
1700027             print_proc_list ();
1700028             print_segments ();
1700029             k_printf (" ");
1700030             print_kmem ();
1700031             k_printf (" ");
1700032             print_time ();
1700033             k_printf ("\n");
1700034             print_mb_map ();
1700035             k_printf ("\n");
1700036             k_exit (0);
1700037           }
1700038         return (pid);
1700039 }
```

100.9.6 kernel/proc/proc_init.c

Si veda la sezione 99.8.6.

```
1710001 #include <kernel/proc.h>
1710002 #include <kernel/k_libc.h>
1710003 #include <string.h>
1710004 //----------------------------------------------------------------------
1710005 extern uint16_t _etext;
1710006 //----------------------------------------------------------------------
1710007 void
1710008 proc_init (void)
1710009 {
1710010     uint8_t  divisor_lo;
1710011     uint8_t  divisor_hi;
1710012     pid_t    pid;
1710013     int      fdn;                  // File descriptor index;
1710014     addr_t   start;                // Used for effective memory addresses.
1710015     size_t   size;                 // Used for memory allocation.
1710016     inode_t *inode;
1710017     sb_t    *sb;
```

```
1710018     //
1710019     // Clear interrupts (should already be cleared).
1710020     //
1710021     cli ();
1710022     //
1710023     // Load Interrupt vector table (IVT).
1710024     //
1710025     ivt_load ();
1710026     //
1710027     // Configure the clock: must be the original values, because
1710028     // the BIOS depends on it!
1710029     //
1710030     // Base frequency is 1193181 Hz and it should divided.
1710031     // Resulting frequency must be from 18.22 Hz and 1193181 Hz.
1710032     // The calculated value (the divisor) must be sent to the
1710033     // PIT (programmable interval timer), divided in two pieces.
1710034     //
1710035     divisor_lo = (CLOCK_FREQUENCY_DIVISOR & 0xFF);          // Low  byte.
1710036     divisor_hi = (CLOCK_FREQUENCY_DIVISOR / 0x100) & 0xFF; // High byte.
1710037     out_8 (0x43, 0x36);
1710038     out_8 (0x40, divisor_lo);        // Lower byte.
1710039     out_8 (0x40, divisor_hi);        // Higher byte.
1710040     //
1710041     // Set all memory reference to some invalid data.
1710042     //
1710043     for (pid = 0; pid < PROCESS_MAX; pid++)
1710044       {
1710045         proc_available (pid);
1710046       }
1710047     //
1710048     // Mount root file system.
1710049     //
1710050     inode = NULL;
1710051     sb = sb_mount (DEV_DSK0, &inode, MOUNT_DEFAULT);
1710052     if (sb == NULL || inode == NULL)
1710053       {
1710054         k_perror ("Kernel panic: cannot mount root file system:");
1710055         k_exit (0);
1710056       }
1710057     //
1710058     // Set up the process table with the kernel.
1710059     //
1710060     proc_table[0].ppid          = 0;
1710061     proc_table[0].pgrp          = 0;
1710062     proc_table[0].uid           = 0;
1710063     proc_table[0].euid          = 0;
1710064     proc_table[0].suid          = 0;
1710065     proc_table[0].device_tty    = DEV_UNDEFINED;
```

```
1710066     proc_table[0].sig_status    = 0;
1710067     proc_table[0].sig_ignore    = 0;
1710068     proc_table[0].usage         = 0;
1710069     proc_table[0].status        = PROC_RUNNING;
1710070     proc_table[0].wakeup_events = 0;
1710071     proc_table[0].wakeup_signal = 0;
1710072     proc_table[0].wakeup_timer  = 0;
1710073     proc_table[0].segment_i     = seg_i ();
1710074     proc_table[0].address_i     = seg_i ();
1710075     proc_table[0].address_i    *= 16;
1710076     proc_table[0].size_i        = (size_t) &_etext;
1710077     proc_table[0].segment_d     = seg_d ();
1710078     proc_table[0].address_d     = seg_d ();
1710079     proc_table[0].address_d    *= 16;
1710080     proc_table[0].size_d        = 0;     // Maximum size: 0x10000.
1710081     proc_table[0].sp            = 0;     // To be set at next interrupt.
1710082     proc_table[0].ret           = 0;
1710083     proc_table[0].umask         = 0022;  // Default umask.
1710084     proc_table[0].inode_cwd     = inode; // Root fs inode.
1710085     strncpy (proc_table[0].path_cwd, "/", PATH_MAX);
1710086     strncpy (proc_table[0].name, "os16 kernel", PATH_MAX);
1710087     //
1710088     // Ensure to have a terminated string.
1710089     //
1710090     proc_table[0].name[PATH_MAX-1] = 0;
1710091     //
1710092     // Reset file descriptors.
1710093     //
1710094     for (fdn = 0; fdn < OPEN_MAX; fdn++)
1710095       {
1710096         proc_table[0].fd[fdn].fl_flags = 0;
1710097         proc_table[0].fd[fdn].fd_flags = 0;
1710098         proc_table[0].fd[fdn].file     = NULL;
1710099       }
1710100     //
1710101     // Allocate memory for the code segment.
1710102     //
1710103     mb_alloc (proc_table[0].address_i, proc_table[0].size_i);
1710104     //
1710105     // Allocate memory for the data segment if different.
1710106     //
1710107     if (seg_d () != seg_i ())
1710108       {
1710109         mb_alloc (proc_table[0].address_d, proc_table[0].size_d);
1710110       }
1710111     //
1710112     // Allocate memory for the BIOS data area (BDA).
1710113     //
```

```
1710114     mb_alloc (0x00000L, 0x500);
1710115     //
1710116     // Allocate memory for the extra BIOS at the
1710117     // bottom of the 640 Kibyte.
1710118     //
1710119     start  = int12 ();
1710120     start *= 1024;
1710121     size   = 0xA0000L - start;
1710122     mb_alloc (start, size);
1710123     //
1710124     // Enable and disable hardware interrupts (IRQ).
1710125     //
1710126     irq_on  (0); // timer.
1710127     irq_on  (1); // enable keyboard
1710128     irq_off (2); //
1710129     irq_off (3); //
1710130     irq_off (4); //
1710131     irq_off (5); //
1710132     irq_on  (6); // floppy (must be on to let int 13 work)!
1710133     irq_off (7); //
1710134     //
1710135     // Interrupts activation.
1710136     //
1710137     sti ();
1710138 }
```

100.9.7 kernel/proc/proc_reference.c

Si veda la sezione 99.8.7.

```
1720001 #include <kernel/proc.h>
1720002 //----------------------------------------------------------------------
1720003 proc_t *
1720004 proc_reference (pid_t pid)
1720005 {
1720006     if (pid >= 0 && pid < PROCESS_MAX)
1720007       {
1720008         return (&proc_table[pid]);
1720009       }
1720010     else
1720011       {
1720012         return (NULL);
1720013       }
1720014 }
```

100.9.8 kernel/proc/proc_sch_signals.c

«

Si veda la sezione 99.8.8.

```
1730001 #include <kernel/proc.h>
1730002 //----------------------------------------------------------------------
1730003 void
1730004 proc_sch_signals (void)
1730005 {
1730006     pid_t  pid;
1730007     for (pid = 0; pid < PROCESS_MAX; pid++)
1730008       {
1730009         proc_sig_term (pid, SIGHUP);
1730010         proc_sig_term (pid, SIGINT);
1730011         proc_sig_core (pid, SIGQUIT);
1730012         proc_sig_core (pid, SIGILL);
1730013         proc_sig_core (pid, SIGABRT);
1730014         proc_sig_core (pid, SIGFPE);
1730015         proc_sig_term (pid, SIGKILL);
1730016         proc_sig_core (pid, SIGSEGV);
1730017         proc_sig_term (pid, SIGPIPE);
1730018         proc_sig_term (pid, SIGALRM);
1730019         proc_sig_term (pid, SIGTERM);
1730020         proc_sig_term (pid, SIGUSR1);
1730021         proc_sig_term (pid, SIGUSR2);
1730022         proc_sig_chld (pid, SIGCHLD);
1730023         proc_sig_cont (pid, SIGCONT);
1730024         proc_sig_stop (pid, SIGSTOP);
1730025         proc_sig_stop (pid, SIGTSTP);
1730026         proc_sig_stop (pid, SIGTTIN);
1730027         proc_sig_stop (pid, SIGTTOU);
1730028       }
1730029 }
```

100.9.9 kernel/proc/proc_sch_terminals.c

«

Si veda la sezione 99.8.9.

```
1740001 #include <kernel/proc.h>
1740002 #include <kernel/k_libc.h>
1740003 //----------------------------------------------------------------------
1740004 void
1740005 proc_sch_terminals (void)
1740006 {
1740007     pid_t  pid;
1740008     int    key;
1740009     tty_t *tty;
1740010     dev_t  device;
1740011     //
```

```
1740012     // Try to read a key from console keyboard buffer (only consoles
1740013     // are available).
1740014     //
1740015     key = con_char_ready ();
1740016     if (key == 0)
1740017       {
1740018         //
1740019         // No key is ready on the keyboard buffer: just return.
1740020         //
1740021         return;
1740022       }
1740023     //
1740024     // A key is available. Find the currently active console.
1740025     //
1740026     device = tty_console ((dev_t) 0);
1740027     tty    = tty_reference (device);
1740028     if (tty == NULL)
1740029       {
1740030         k_printf ("kernel alert: console device 0x%04x not found!\n",
1740031                   device);
1740032         //
1740033         // Will send the typed character to the first terminal!
1740034         //
1740035         tty = tty_reference ((dev_t) 0);
1740036       }
1740037     //
1740038     // Defined the active console. Put the character there.
1740039     //
1740040     if (tty->key == 0)
1740041       {
1740042         tty->status = TTY_OK;
1740043       }
1740044     else
1740045       {
1740046         tty->status = TTY_LOST_KEY;
1740047       }
1740048     tty->key = con_char_read ();
1740049     //
1740050     // Verify if it is a control key that must be handled. If so, a
1740051     // signal is sent to all processes with the same control terminal,
1740052     // excluded the kernel (0) and ‘init’ (1). Such control keys are not
1740053     // passed to the applications.
1740054     //
1740055     // Please note that this a simplified solution, because the signal
1740056     // should reach only the foreground process of the group. For that
1740057     // reason, only che [Ctrl C] is taken into consideration, because
1740058     // processes can ignore the signal ’SIGINT’.
1740059     //
```

```
1740060     if (tty->pgrp != 0)
1740061       {
1740062         //
1740063         // There is a process group for that terminal.
1740064         //
1740065         if      (tty->key == 3)         // [Ctrl C] -> SIGINT
1740066           {
1740067             for (pid = 2; pid <  PROCESS_MAX; pid++)
1740068               {
1740069                 if (proc_table[pid].pgrp == tty->pgrp)
1740070                   {
1740071                     k_kill (pid, SIGINT);
1740072                   }
1740073               }
1740074             tty->key    = 0;    // Reset key and status.
1740075             tty->status = TTY_OK;
1740076           }
1740077       }
1740078     //
1740079     // Check for a console switch key combination.
1740080     //
1740081     if      (tty->key == 0x11)          // [Ctrl Q] -> DC1 -> console0.
1740082       {
1740083         tty->key    = 0;                // Reset key and status.
1740084         tty->status = TTY_OK;
1740085         tty_console (DEV_CONSOLE0);     // Switch.
1740086       }
1740087     else if (tty->key == 0x12)          // [Ctrl R] -> DC2 -> console1.
1740088       {
1740089         tty->key    = 0;                // Reset key and status.
1740090         tty->status = TTY_OK;
1740091         tty_console (DEV_CONSOLE1);     // Switch.
1740092       }
1740093     else if (tty->key == 0x13)          // [Ctrl S] -> DC3 -> console2.
1740094       {
1740095         tty->key    = 0;                // Reset key and status.
1740096         tty->status = TTY_OK;
1740097         tty_console (DEV_CONSOLE2);     // Switch.
1740098       }
1740099     else if (tty->key == 0x14)          // [Ctrl T] -> DC4 -> console3.
1740100       {
1740101         tty->key    = 0;                // Reset key and status.
1740102         tty->status = TTY_OK;
1740103         tty_console (DEV_CONSOLE3);     // Switch.
1740104       }
1740105     //
1740106     // A key was pressed: must wake up all processes waiting for reading
1740107     // a terminal: all processes must be reactivated, because a process
```

```
1740108     // can read from the device file, and not just from its own
1740109     // terminal.
1740110     //
1740111     for (pid = 0; pid < PROCESS_MAX; pid++)
1740112       {
1740113         if (    (proc_table[pid].status == PROC_SLEEPING)
1740114             && (proc_table[pid].wakeup_events & WAKEUP_EVENT_TTY))
1740115           {
1740116             //
1740117             // A process waiting for that terminal was found:
1740118             // remove the waiting event and set it ready.
1740119             //
1740120             proc_table[pid].wakeup_events &= ~WAKEUP_EVENT_TTY;
1740121             proc_table[pid].status = PROC_READY;
1740122           }
1740123       }
1740124 }
```

100.9.10 kernel/proc/proc_sch_timers.c

Si veda la sezione 99.8.10.

```
1750001 #include <kernel/proc.h>
1750002 #include <kernel/k_libc.h>
1750003 //----------------------------------------------------------------------
1750004 void
1750005 proc_sch_timers (void)
1750006 {
1750007     static unsigned long int previous_time;
1750008            unsigned long int current_time;
1750009            unsigned int      pid;
1750010     current_time = k_time (NULL);
1750011     if (previous_time != current_time)
1750012       {
1750013         for (pid = 0; pid < PROCESS_MAX; pid++)
1750014           {
1750015             if (    (proc_table[pid].wakeup_events & WAKEUP_EVENT_TIMER)
1750016                 && (proc_table[pid].status == PROC_SLEEPING)
1750017                 && (proc_table[pid].wakeup_timer > 0))
1750018               {
1750019                 proc_table[pid].wakeup_timer--;
1750020                 if (proc_table[pid].wakeup_timer == 0)
1750021                   {
1750022                     proc_table[pid].status = PROC_READY;
1750023                   }
1750024               }
1750025           }
1750026       }
```

```
1750027     previous_time = current_time;
1750028 }
```

100.9.11 kernel/proc/proc_scheduler.c

«

Si veda la sezione 99.8.11.

```
1760001 #include <kernel/proc.h>
1760002 #include <kernel/k_libc.h>
1760003 #include <stdint.h>
1760004 //----------------------------------------------------------------------
1760005 extern uint16_t _ksp;
1760006 //----------------------------------------------------------------------
1760007 void
1760008 proc_scheduler (uint16_t *sp, segment_t *segment_d)
1760009 {
1760010     //
1760011     // The process is identified from the data and stack segment.
1760012     //
1760013     pid_t prev;
1760014     pid_t next;
1760015     //
1760016     static unsigned long int previous_clock;
1760017            unsigned long int current_clock;
1760018     //
1760019     // Check if current data segments are right.
1760020     //
1760021     if (es () != ds () || ss () != ds ())
1760022       {
1760023         k_printf ("\n");
1760024         k_printf ("Kernel panic: ES, DS, SS are different!\n");
1760025         k_exit (0);
1760026       }
1760027     //
1760028     // Search the data segment inside the process table.
1760029     // Must be done here, because the subsequent call to
1760030     // proc_sch_signals() will remove the segment numbers
1760031     // from a zombie process.
1760032     //
1760033     prev = proc_find (*segment_d);
1760034     //
1760035     // Take care of sleeping processes: wake up if sleeping time
1760036     // elapsed.
1760037     //
1760038     proc_sch_timers ();
1760039     //
1760040     // Take care of pending signals.
1760041     //
```

```
1760042      proc_sch_signals ();
1760043      //
1760044      // Take care input from terminals.
1760045      //
1760046      proc_sch_terminals ();
1760047      //
1760048      // Update the CPU time usage.
1760049      //
1760050      current_clock   = k_clock ();
1760051      proc_table[prev].usage += current_clock - previous_clock;
1760052      previous_clock  = current_clock;
1760053      //
1760054      // Scan for a next process.
1760055      //
1760056      for (next = prev+1; next != prev; next++)
1760057        {
1760058          if (next >= PROCESS_MAX)
1760059            {
1760060              next = -1; // At the next loop, 'next' will be zero.
1760061              continue;
1760062            }
1760063          if      (proc_table[next].status == PROC_EMPTY)
1760064            {
1760065              continue;
1760066            }
1760067          else if (proc_table[next].status == PROC_CREATED)
1760068            {
1760069              continue;
1760070            }
1760071          else if (proc_table[next].status == PROC_READY)
1760072            {
1760073              if (proc_table[prev].status == PROC_RUNNING)
1760074                {
1760075                  proc_table[prev].status = PROC_READY;
1760076                }
1760077              proc_table[prev].sp     = *sp;
1760078              proc_table[next].status = PROC_RUNNING;
1760079              proc_table[next].ret    = 0;
1760080              *segment_d      = proc_table[next].segment_d;
1760081              *sp             = proc_table[next].sp;
1760082              break;
1760083            }
1760084          else if (proc_table[next].status == PROC_RUNNING)
1760085            {
1760086              if (proc_table[prev].status == PROC_RUNNING)
1760087                {
1760088                  k_printf ("Kernel alert: process %i ", prev);
1760089                  k_printf ("and %i \"running\"!\r", next);
```

```
1760090                     proc_table[prev].status = PROC_READY;
1760091                   }
1760092                 proc_table[prev].sp      = *sp;
1760093                 proc_table[next].status = PROC_RUNNING;
1760094                 proc_table[next].ret     = 0;
1760095                 *segment_d       = proc_table[next].segment_d;
1760096                 *sp              = proc_table[next].sp;
1760097                 break;
1760098               }
1760099             else if (proc_table[next].status == PROC_SLEEPING)
1760100               {
1760101                 continue;
1760102               }
1760103             else if (proc_table[next].status == PROC_ZOMBIE)
1760104               {
1760105                 continue;
1760106               }
1760107           }
1760108         //
1760109         // Check again if the next process is set to running, otherwise set
1760110         // the kernel to such value!
1760111         //
1760112         next = proc_find (*segment_d);
1760113         if (proc_table[next].status != PROC_RUNNING)
1760114           {
1760115             proc_table[0].status = PROC_RUNNING;
1760116             *segment_d    = proc_table[0].segment_d;
1760117             *sp           = proc_table[0].sp;
1760118           }
1760119         //
1760120         // Save kernel stack pointer.
1760121         //
1760122         _ksp = proc_table[0].sp;
1760123         //
1760124         // At the end, must inform the PIC 1, with message «EOI».
1760125         //
1760126         out_8 (0x20, 0x20);
1760127     }
```

100.9.12 kernel/proc/proc_sig_chld.c

Si veda la sezione 99.8.12.

```
1770001     #include <kernel/proc.h>
1770002     //----------------------------------------------------------------------
1770003     void
1770004     proc_sig_chld (pid_t parent, int sig)
1770005     {
```

```
1770006     pid_t child;
1770007     //
1770008     // Please note that ‘sig’ should be SIGCHLD and nothing else.
1770009     // So, the following test, means to verify if the parent process
1770010     // has received a SIGCHLD already.
1770011     //
1770012     if (proc_sig_status (parent, sig))
1770013       {
1770014         if (   (!proc_sig_ignore (parent, sig))
1770015             && (proc_table[parent].status == PROC_SLEEPING)
1770016             && (proc_table[parent].wakeup_events & WAKEUP_EVENT_SIGNAL)
1770017             && (proc_table[parent].wakeup_signal == sig))
1770018           {
1770019             //
1770020             // The signal is not ignored from the parent process;
1770021             // the parent process is sleeping;
1770022             // the parent process is waiting for a signal;
1770023             // the parent process is waiting for current signal.
1770024             // So, just wake it up.
1770025             //
1770026             proc_table[parent].status         = PROC_READY;
1770027             proc_table[parent].wakeup_events ^= WAKEUP_EVENT_SIGNAL;
1770028             proc_table[parent].wakeup_signal  = 0;
1770029           }
1770030         else
1770031           {
1770032             //
1770033             // All other cases, means to remove all dead children.
1770034             //
1770035             for (child = 1; child < PROCESS_MAX; child++)
1770036               {
1770037                 if (   proc_table[child].ppid == parent
1770038                     && proc_table[child].status == PROC_ZOMBIE)
1770039                   {
1770040                     proc_available (child);
1770041                   }
1770042               }
1770043           }
1770044         proc_sig_off (parent, sig);
1770045       }
1770046 }
```

100.9.13 kernel/proc/proc_sig_cont.c

«

Si veda la sezione 99.8.13.

```
1780001 #include <kernel/proc.h>
1780002 //----------------------------------------------------------------------
1780003 void
1780004 proc_sig_cont (pid_t pid, int sig)
1780005 {
1780006     //
1780007     // The value for argument 'sig' should be SIGCONT.
1780008     //
1780009     if (proc_sig_status (pid, sig))
1780010       {
1780011         if (proc_sig_ignore (pid, sig))
1780012           {
1780013             proc_sig_off (pid, sig);
1780014           }
1780015         else
1780016           {
1780017             proc_table[pid].status = PROC_READY;
1780018             proc_sig_off (pid, sig);
1780019           }
1780020       }
1780021 }
```

100.9.14 kernel/proc/proc_sig_core.c

«

Si veda la sezione 99.8.14.

```
1790001 #include <kernel/proc.h>
1790002 //----------------------------------------------------------------------
1790003 void
1790004 proc_sig_core (pid_t pid, int sig)
1790005 {
1790006     addr_t address_i;
1790007     addr_t address_d;
1790008     size_t size_i;
1790009     size_t size_d;
1790010     //
1790011     if (proc_sig_status (pid, sig))
1790012       {
1790013         if (proc_sig_ignore (pid, sig))
1790014           {
1790015             proc_sig_off (pid, sig);
1790016           }
1790017         else
1790018           {
1790019             //
```

```
1790020                 // Save process addresses and sizes (might be useful if
1790021                 // we want to try to exit the process before core dump.
1790022                 //
1790023                 address_i   = proc_table[pid].address_i;
1790024                 address_d   = proc_table[pid].address_d;
1790025                 size_i      = proc_table[pid].size_i;
1790026                 size_d      = proc_table[pid].size_d;
1790027                 //
1790028                 // Core dump: the process who formally writes the file
1790029                 // is the terminating one.
1790030                 //
1790031                 if (address_d == address_i)
1790032                   {
1790033                     proc_dump_memory (pid, address_i, size_i, "core");
1790034                   }
1790035                 else
1790036                   {
1790037                     proc_dump_memory (pid, address_i, size_i, "core.i");
1790038                     proc_dump_memory (pid, address_d, size_d, "core.d");
1790039                   }
1790040                 //
1790041                 // The signal, translated to negative, is returned (but
1790042                 // the effective value received by the application will
1790043                 // be cutted, leaving only the low 8 bit).
1790044                 //
1790045                 proc_sys_exit (pid, -sig);
1790046               }
1790047           }
1790048 }
```

100.9.15 kernel/proc/proc_sig_ignore.c

Si veda la sezione 99.8.15.

```
1800001 #include <kernel/proc.h>
1800002 //----------------------------------------------------------------------
1800003 int
1800004 proc_sig_ignore (pid_t pid, int sig)
1800005 {
1800006     unsigned long int flag = 1L << (sig - 1);
1800007     if (proc_table[pid].sig_ignore & flag)
1800008       {
1800009         return (1);
1800010       }
1800011     else
1800012       {
1800013         return (0);
1800014       }
```

```
1800015 }
```

100.9.16 kernel/proc/proc_sig_off.c

Si veda la sezione 99.8.16.

```
1810001 #include <kernel/proc.h>
1810002 //----------------------------------------------------------------------
1810003 void
1810004 proc_sig_off (pid_t pid, int sig)
1810005 {
1810006     unsigned long int flag = 1L << (sig - 1);
1810007     proc_table[pid].sig_status ^= flag;
1810008 }
```

100.9.17 kernel/proc/proc_sig_on.c

Si veda la sezione 99.8.16.

```
1820001 #include <kernel/proc.h>
1820002 //----------------------------------------------------------------------
1820003 void
1820004 proc_sig_on (pid_t pid, int sig)
1820005 {
1820006     unsigned long int flag = 1L << (sig - 1);
1820007     proc_table[pid].sig_status |= flag;
1820008 }
```

100.9.18 kernel/proc/proc_sig_status.c

Si veda la sezione 99.8.17.

```
1830001 #include <kernel/proc.h>
1830002 //----------------------------------------------------------------------
1830003 int
1830004 proc_sig_status (pid_t pid, int sig)
1830005 {
1830006     unsigned long int flag = 1L << (sig - 1);
1830007     if (proc_table[pid].sig_status & flag)
1830008       {
1830009         return (1);
1830010       }
1830011     else
1830012       {
1830013         return (0);
1830014       }
1830015 }
```

100.9.19 kernel/proc/proc_sig_stop.c

Si veda la sezione 99.8.18.

```
1840001 #include <kernel/proc.h>
1840002 //----------------------------------------------------------------------
1840003 void
1840004 proc_sig_stop (pid_t pid, int sig)
1840005 {
1840006     if (proc_sig_status (pid, sig))
1840007       {
1840008         if (proc_sig_ignore (pid, sig) && !(sig == SIGSTOP))
1840009           {
1840010             proc_sig_off (pid, sig);
1840011           }
1840012         else
1840013           {
1840014             proc_table[pid].status = PROC_SLEEPING;
1840015             proc_table[pid].ret    = -sig;
1840016             proc_sig_off (pid, sig);
1840017           }
1840018       }
1840019 }
```

100.9.20 kernel/proc/proc_sig_term.c

Si veda la sezione 99.8.19.

```
1850001 #include <kernel/proc.h>
1850002 //----------------------------------------------------------------------
1850003 void
1850004 proc_sig_term (pid_t pid, int sig)
1850005 {
1850006     if (proc_sig_status (pid, sig))
1850007       {
1850008         if (proc_sig_ignore (pid, sig) && !(sig == SIGKILL))
1850009           {
1850010             proc_sig_off (pid, sig);
1850011           }
1850012         else
1850013           {
1850014             //
1850015             // The signal, translated to negative, is returned (but
1850016             // the effective value received by the application will
1850017             // be cutted, leaving only the low 8 bit).
1850018             //
1850019             proc_sys_exit (pid, -sig);
1850020           }
1850021       }
```

```
1850022 }
```

100.9.21 kernel/proc/proc_sys_exec.c

«

Si veda la sezione 99.8.20.

```
1860001 #include <kernel/proc.h>
1860002 #include <errno.h>
1860003 #include <fcntl.h>
1860004 //----------------------------------------------------------------------
1860005 int
1860006 proc_sys_exec (uint16_t *sp, segment_t *segment_d, pid_t pid,
1860007                const char *path,
1860008                unsigned int argc, char *arg_data,
1860009                unsigned int envc, char *env_data)
1860010 {
1860011   unsigned int      i;
1860012   unsigned int      j;
1860013   char             *arg;
1860014   char             *env;
1860015   char             *envp[ARG_MAX/16];
1860016   char             *argv[ARG_MAX/16];
1860017   size_t            size;
1860018   size_t            arg_data_size;
1860019   size_t            env_data_size;
1860020   unsigned int      p_off;
1860021   inode_t          *inode;
1860022   ssize_t           size_read;
1860023   header_t          header;
1860024   unsigned long int s;          // Help calculating process sizes.
1860025   uint16_t          new_sp;
1860026   uint16_t          envp_address;
1860027   uint16_t          argv_address;
1860028   size_t            process_size_i;
1860029   size_t            process_size_d;
1860030   char              buffer[MEM_BLOCK_SIZE];
1860031   uint16_t          stack_element;
1860032   off_t             inode_start;
1860033   addr_t            memory_start;
1860034   addr_t            previous_address_i;
1860035   segment_t         previous_segment_i;
1860036   size_t            previous_size_i;
1860037   addr_t            previous_address_d;
1860038   segment_t         previous_segment_d;
1860039   size_t            previous_size_d;
1860040   int               status;
1860041   memory_t          allocated_i;
1860042   memory_t          allocated_d;
```

```
1860043     pid_t              extra;
1860044     int                proc_count;
1860045     file_t            *file;
1860046     int                fdn;
1860047     dev_t              device;
1860048     int                eof;
1860049     //
1860050     // Check for limits.
1860051     //
1860052     if (argc > (ARG_MAX/16) || envc > (ARG_MAX/16))
1860053       {
1860054         errset (ENOMEM);
1860055         return (-1);
1860056       }
1860057     //
1860058     // Scan arguments to calculate the full size and the relative
1860059     // pointers. The final size is rounded to 2, for the stack.
1860060     //
1860061     arg = arg_data;
1860062     for (i = 0, j = 0; i < argc; i++)
1860063       {
1860064         argv[i] = (char *) j;   // Relative pointer inside
1860065                                 // the 'arg_data'.
1860066         size = strlen (arg);
1860067         arg += size + 1;
1860068         j += size + 1;
1860069       }
1860070     arg_data_size = j;
1860071     if (arg_data_size % 2)
1860072       {
1860073         arg_data_size++;
1860074       }
1860075     //
1860076     // Scan environment variables to calculate the full size and the
1860077     // relative pointers. The final size is rounded to 2, for the stack.
1860078     //
1860079     env = env_data;
1860080     for (i = 0, j = 0; i < envc; i++)
1860081       {
1860082         envp[i] = (char *) j;   // Relative pointer inside
1860083                                 // the 'env_data'.
1860084         size = strlen (env);
1860085         env += size + 1;
1860086         j += size + 1;
1860087       }
1860088     env_data_size = j;
1860089     if (env_data_size % 2)
1860090       {
```

```
1860091             env_data_size++;
1860092           }
1860093         //
1860094         // Read the inode related to the executable file name.
1860095         // Function path_inode() includes the inode get procedure.
1860096         //
1860097         inode = path_inode (pid, path);
1860098         if (inode == NULL)
1860099           {
1860100             errset (ENOENT);         // No such file or directory.
1860101             return (-1);
1860102           }
1860103         //
1860104         // Check for permissions.
1860105         //
1860106         status = inode_check (inode, S_IFREG, 5, proc_table[pid].euid);
1860107         if (status != 0)
1860108           {
1860109             //
1860110             // File is not of a valid type or permission are not
1860111             // sufficient: release the executable file inode
1860112             // and return with an error.
1860113             //
1860114             inode_put (inode);
1860115             errset (EACCES);         // Permission denied.
1860116             return (-1);
1860117           }
1860118         //
1860119         // Read the header from the executable file.
1860120         //
1860121         size_read = inode_file_read (inode, (off_t) 0, &header,
1860122                                      (sizeof header), &eof);
1860123         if (size_read != (sizeof header))
1860124           {
1860125             //
1860126             // The file is shorter than the executable header, so, it isn't
1860127             // an executable: release the file inode and return with an
1860128             // error.
1860129             //
1860130             inode_put (inode);
1860131             errset (ENOEXEC);
1860132             return (-1);
1860133           }
1860134         if (header.magic0 != MAGIC_OS16 || header.magic1 != MAGIC_OS16_APPL)
1860135           {
1860136             //
1860137             // The header does not have the expected magic numbers, so,
1860138             // it isn't a valid executable: release the file inode and
```

```
1860139         // return with an error.
1860140         //
1860141         inode_put (inode);
1860142         errset (ENOEXEC);
1860143         return (-1);    // This is not a valid executable!
1860144       }
1860145     //
1860146     // Calculate data size.
1860147     //
1860148     s  = header.ebss;
1860149     if (header.ssize == 0)
1860150       {
1860151         s += 0x10000L;  // Zero means max size.
1860152       }
1860153     else
1860154       {
1860155         s += header.ssize;
1860156       }
1860157     if (s > 0xFFFF)
1860158       {
1860159         process_size_d = 0x0000;  // 0x0000 means the maximum size:
1860160                                   // 0x10000.
1860161         new_sp         = 0x0000;  // 0x0000 is like 0x10000 and the first
1860162                                   // push moves SP to 0xFFFE.
1860163       }
1860164     else
1860165       {
1860166         process_size_d = s;
1860167         new_sp         = process_size_d;
1860168         if (new_sp % 2)
1860169           {
1860170             new_sp--;        // The stack pointer should be even.
1860171           }
1860172       }
1860173     //
1860174     // Calculate code size.
1860175     //
1860176     if (header.segoff == 0)
1860177       {
1860178         process_size_i = process_size_d;
1860179       }
1860180     else
1860181       {
1860182         process_size_i = header.segoff * 16;
1860183       }
1860184     //
1860185     // Allocate memory: code and data segments.
1860186     //
```

```
1860187     status = mb_alloc_size (process_size_i, &allocated_i);
1860188     if (status < 0)
1860189       {
1860190         //
1860191         // The program instructions (code segment) cannot be loaded
1860192         // into memory: release the executable file inode and return
1860193         // with an error.
1860194         //
1860195         inode_put (inode);
1860196         errset (ENOMEM);        // Not enough space.
1860197         return ((pid_t) -1);
1860198       }
1860199     if (header.segoff == 0)
1860200       {
1860201         //
1860202         // Code and data segments are the same: no need
1860203         // to allocate more memory for the data segment.
1860204         //
1860205         allocated_d.address = allocated_i.address;
1860206         allocated_d.segment = allocated_i.segment;
1860207         allocated_d.size    = allocated_i.size;
1860208       }
1860209     else
1860210       {
1860211         //
1860212         // Code and data segments are different: the data
1860213         // segment memory is allocated.
1860214         //
1860215         status = mb_alloc_size (process_size_d, &allocated_d);
1860216         if (status < 0)
1860217           {
1860218             //
1860219             // The separated program data (data segment) cannot be loaded
1860220             // into memory: free the already allocated memory for the
1860221             // program instructions, release the executable file inode
1860222             // and return with an error.
1860223             //
1860224             mb_free (allocated_i.address, allocated_i.size);
1860225             inode_put (inode);
1860226             errset (ENOMEM);        // Not enough space.
1860227             return ((pid_t) -1);
1860228           }
1860229       }
1860230     //
1860231     // Load executable in memory.
1860232     //
1860233     if (header.segoff == 0)
1860234       {
```

```
1860235         //
1860236         // Code and data share the same segment.
1860237         //
1860238         for (eof = 0, memory_start = allocated_i.address,
1860239              inode_start = 0, size_read = 0;
1860240              inode_start < inode->size && !eof;
1860241              inode_start += size_read)
1860242           {
1860243             memory_start += size_read;
1860244             //
1860245             // Read a block of memory.
1860246             //
1860247             size_read = inode_file_read (inode, inode_start,
1860248                                          buffer, MEM_BLOCK_SIZE, &eof);
1860249             if (size_read < 0)
1860250               {
1860251                 //
1860252                 // Free memory and inode.
1860253                 //
1860254                 mb_free (allocated_i.address, allocated_i.size);
1860255                 inode_put (inode);
1860256                 errset (EIO);
1860257                 return (-1);
1860258               }
1860259             //
1860260             // Copy inside the right position to be executed.
1860261             //
1860262             dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
1860263                     (size_t) size_read, NULL);
1860264           }
1860265       }
1860266     else
1860267       {
1860268         //
1860269         // Code and data with different segments.
1860270         //
1860271         for (eof = 0, memory_start = allocated_i.address,
1860272              inode_start = 0, size_read = 0;
1860273              inode_start < process_size_i && !eof;
1860274              inode_start += size_read)
1860275           {
1860276             memory_start += size_read;
1860277             //
1860278             // Read a block of memory
1860279             //
1860280             size_read = inode_file_read (inode, inode_start,
1860281                                          buffer, MEM_BLOCK_SIZE, &eof);
1860282             if (size_read < 0)
```

```
1860283              {
1860284                //
1860285                // Free memory and inode.
1860286                //
1860287                mb_free (allocated_i.address, allocated_i.size);
1860288                mb_free (allocated_d.address, allocated_d.size);
1860289                inode_put (inode);
1860290                errset (EIO);
1860291                return (-1);
1860292              }
1860293            //
1860294            // Copy inside the right position to be executed.
1860295            //
1860296            dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
1860297                    (size_t) size_read, NULL);
1860298          }
1860299        for (eof = 0, memory_start = allocated_d.address,
1860300             inode_start = (header.segoff * 16), size_read = 0;
1860301             inode_start < inode->size && !eof;
1860302             inode_start += size_read)
1860303          {
1860304            memory_start += size_read;
1860305            //
1860306            // Read a block of memory
1860307            //
1860308            size_read = inode_file_read (inode, inode_start,
1860309                                         buffer, MEM_BLOCK_SIZE, &eof);
1860310            if (size_read < 0)
1860311              {
1860312                //
1860313                // Free memory and inode.
1860314                //
1860315                mb_free (allocated_i.address, allocated_i.size);
1860316                mb_free (allocated_d.address, allocated_d.size);
1860317                inode_put (inode);
1860318                errset (EIO);
1860319                return (-1);
1860320              }
1860321            dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
1860322                    (size_t) size_read, NULL);
1860323          }
1860324      }
1860325    //
1860326    // The executable file was successfully loaded in memory:
1860327    // release the executable file inode.
1860328    //
1860329    inode_put (inode);
1860330    //
```

```
1860331     // Put environment data inside the stack.
1860332     //
1860333     new_sp -= env_data_size; //------------------------------ environment
1860334     mem_copy (address (seg_d (), (unsigned int) env_data),
1860335               (allocated_d.address + new_sp), env_data_size);
1860336     //
1860337     // Put arguments data inside the stack.
1860338     //
1860339     new_sp -= arg_data_size; //-------------------------------- arguments
1860340     mem_copy (address (seg_d (), (unsigned int) arg_data),
1860341               (allocated_d.address + new_sp), arg_data_size);
1860342     //
1860343     // Put envp[] inside the stack, updating all the pointers.
1860344     //
1860345     new_sp -= 2; //--------------------------------------------------- NULL
1860346     stack_element = NULL;
1860347     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860348             (allocated_d.address + new_sp),
1860349             &stack_element, (sizeof stack_element), NULL);
1860350     //
1860351     p_off  = new_sp;            //
1860352     p_off += 2;                 // Calculate memory pointers from
1860353     p_off += arg_data_size;     // original relative pointers,
1860354     for (i = 0; i < envc; i++)  // inside the environment array
1860355       {                         // of pointers.
1860356         envp[i] += p_off;       //
1860357       }                         //
1860358     //
1860359     new_sp -= (envc * (sizeof (char *))); //--------------------- *envp[]
1860360     mem_copy (address (seg_d (), (unsigned int) envp),
1860361               (allocated_d.address + new_sp),
1860362               (envc * (sizeof (char *))));
1860363     //
1860364     // Save the envp[] location, needed in the following.
1860365     //
1860366     envp_address = new_sp;
1860367     //
1860368     // Put argv[] inside the stack, updating all the pointers.
1860369     //
1860370     new_sp -= 2; //--------------------------------------------------- NULL
1860371     stack_element = NULL;
1860372     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860373             (allocated_d.address + new_sp),
1860374             &stack_element, (sizeof stack_element), NULL);
1860375     //
1860376     p_off  = new_sp;                      //
1860377     p_off += 2 ;                          // Calculate memory pointers
1860378     p_off += (envc * (sizeof (char *)));  // from original relative
```

```
1860379     p_off += 2;                            // pointers, inside the
1860380     for (i = 0; i < argc; i++)             // arguments array of
1860381       {                                    // pointers.
1860382         argv[i] += p_off;                  //
1860383       }                                    //
1860384     //
1860385     new_sp -= (argc * (sizeof (char *))); //---------------------- *argv[]
1860386     mem_copy (address (seg_d (), (unsigned int) argv),
1860387               (allocated_d.address + new_sp),
1860388               (argc * (sizeof (char *))));
1860389     //
1860390     // Save the argv[] location, needed in the following.
1860391     //
1860392     argv_address = new_sp;
1860393     //
1860394     // Put the pointer to the array envp[].
1860395     //
1860396     new_sp -= 2; //---------------------------------------------------- argc
1860397     stack_element = envp_address;
1860398     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860399             (allocated_d.address + new_sp),
1860400             &stack_element, (sizeof stack_element), NULL);
1860401     //
1860402     // Put the pointer to the array argv[].
1860403     //
1860404     new_sp -= 2; //---------------------------------------------------- argc
1860405     stack_element = argv_address;
1860406     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860407             (allocated_d.address + new_sp),
1860408             &stack_element, (sizeof stack_element), NULL);
1860409     //
1860410     // Put argc inside the stack.
1860411     //
1860412     new_sp -= 2; //---------------------------------------------------- argc
1860413     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860414             (allocated_d.address + new_sp),
1860415             &argc, (sizeof argc), NULL);
1860416     //
1860417     // Set the rest of the stack.
1860418     //
1860419     new_sp -= 2; //--------------------------------------------------- FLAGS
1860420     stack_element = 0x0200;
1860421     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860422             (allocated_d.address + new_sp),
1860423             &stack_element, (sizeof stack_element), NULL);
1860424     new_sp -= 2; //------------------------------------------------------ CS
1860425     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860426             (allocated_d.address + new_sp),
```

```
1860427             &allocated_i.segment, (sizeof allocated_i.segment), NULL);
1860428     new_sp -= 2; //------------------------------------------------------ IP
1860429     stack_element = 0;
1860430     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860431             (allocated_d.address + new_sp),
1860432             &stack_element, (sizeof stack_element), NULL);
1860433     new_sp -= 2; //------------------------------------------------------ ES
1860434     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860435             (allocated_d.address + new_sp),
1860436             &allocated_d.segment, (sizeof allocated_d.segment), NULL);
1860437     new_sp -= 2; //------------------------------------------------------ DS
1860438     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860439             (allocated_d.address + new_sp),
1860440             &allocated_d.segment, (sizeof allocated_d.segment), NULL);
1860441     new_sp -= 2; //------------------------------------------------------ DI
1860442     stack_element = 0;
1860443     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860444             (allocated_d.address + new_sp),
1860445             &stack_element, (sizeof stack_element), NULL);
1860446     new_sp -= 2; //------------------------------------------------------ SI
1860447     stack_element = 0;
1860448     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860449             (allocated_d.address + new_sp),
1860450             &stack_element, (sizeof stack_element), NULL);
1860451     new_sp -= 2; //------------------------------------------------------ BP
1860452     stack_element = 0;
1860453     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860454             (allocated_d.address + new_sp),
1860455             &stack_element, (sizeof stack_element), NULL);
1860456     new_sp -= 2; //------------------------------------------------------ BX
1860457     stack_element = 0;
1860458     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860459             (allocated_d.address + new_sp),
1860460             &stack_element, (sizeof stack_element), NULL);
1860461     new_sp -= 2; //------------------------------------------------------ DX
1860462     stack_element = 0;
1860463     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860464             (allocated_d.address + new_sp),
1860465             &stack_element, (sizeof stack_element), NULL);
1860466     new_sp -= 2; //------------------------------------------------------ CX
1860467     stack_element = 0;
1860468     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860469             (allocated_d.address + new_sp),
1860470             &stack_element, (sizeof stack_element), NULL);
1860471     new_sp -= 2; //------------------------------------------------------ AX
1860472     stack_element = 0;
1860473     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860474             (allocated_d.address + new_sp),
```

```
1860475             &stack_element, (sizeof stack_element), NULL);
1860476     //
1860477     // Close process file descriptors, if the 'FD_CLOEXEC' flag
1860478     // is present.
1860479     //
1860480     for (fdn = 0; fdn < OPEN_MAX; fdn++)
1860481       {
1860482         if (proc_table[pid].fd[0].file != NULL)
1860483           {
1860484             if (proc_table[pid].fd[0].fd_flags & FD_CLOEXEC)
1860485               {
1860486                 fd_close (pid, fdn);
1860487               }
1860488           }
1860489       }
1860490     //
1860491     // Select device for standard I/O, if a standard I/O stream must be
1860492     // opened.
1860493     //
1860494     if (proc_table[pid].device_tty != 0)
1860495       {
1860496         device = proc_table[pid].device_tty;
1860497       }
1860498     else
1860499       {
1860500         device = DEV_TTY;
1860501       }
1860502     //
1860503     // Prepare missing standard file descriptors. The function
1860504     // 'file_stdio_dev_make()' arranges the value for 'errno' if
1860505     // necessary. If a standard file descriptor cannot be allocated,
1860506     // the program is left without it.
1860507     //
1860508     if (proc_table[pid].fd[0].file == NULL)
1860509       {
1860510         file = file_stdio_dev_make (device, S_IFCHR, O_RDONLY);
1860511         if (file != NULL)                                      // stdin
1860512           {
1860513             proc_table[pid].fd[0].fl_flags     = O_RDONLY;
1860514             proc_table[pid].fd[0].fd_flags     = 0;
1860515             proc_table[pid].fd[0].file         = file;
1860516             proc_table[pid].fd[0].file->offset = 0;
1860517           }
1860518       }
1860519     if (proc_table[pid].fd[1].file == NULL)
1860520       {
1860521         file = file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
1860522         if (file != NULL)                                     // stdout
```

```
1860523             {
1860524               proc_table[pid].fd[1].fl_flags     = O_WRONLY;
1860525               proc_table[pid].fd[1].fd_flags     = 0;
1860526               proc_table[pid].fd[1].file         = file;
1860527               proc_table[pid].fd[1].file->offset = 0;
1860528             }
1860529         }
1860530       if (proc_table[pid].fd[2].file == NULL)
1860531         {
1860532           file = file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
1860533           if (file != NULL)                                    // stderr
1860534             {
1860535               proc_table[pid].fd[2].fl_flags     = O_WRONLY;
1860536               proc_table[pid].fd[2].fd_flags     = 0;
1860537               proc_table[pid].fd[2].file         = file;
1860538               proc_table[pid].fd[2].file->offset = 0;
1860539             }
1860540         }
1860541       //
1860542       // Prepare to switch
1860543       //
1860544       previous_address_i = proc_table[pid].address_i;
1860545       previous_segment_i = proc_table[pid].segment_i;
1860546       previous_size_i    = proc_table[pid].size_i;
1860547       previous_address_d = proc_table[pid].address_d;
1860548       previous_segment_d = proc_table[pid].segment_d;
1860549       previous_size_d    = proc_table[pid].size_d;
1860550       //
1860551       proc_table[pid].address_i  = allocated_i.address;
1860552       proc_table[pid].segment_i  = allocated_i.segment;
1860553       proc_table[pid].size_i     = allocated_i.size;
1860554       proc_table[pid].address_d  = allocated_d.address;
1860555       proc_table[pid].segment_d  = allocated_d.segment;
1860556       proc_table[pid].size_d     = allocated_d.size;
1860557       proc_table[pid].sp         = new_sp;
1860558       strncpy (proc_table[pid].name, path, PATH_MAX);
1860559       //
1860560       // Ensure to have a terminated string.
1860561       //
1860562       proc_table[pid].name[PATH_MAX-1] = 0;
1860563       //
1860564       // Free data segment memory.
1860565       //
1860566       mb_free (previous_address_d, previous_size_d);
1860567       //
1860568       // Free code segment memory if it is
1860569       // different from the data segment.
1860570       //
```

```
1860571   if (previous_segment_i != previous_segment_d)
1860572     {
1860573       //
1860574       // Must verify if no other process is
1860575       // using the same memory.
1860576       //
1860577       for (proc_count = 0, extra = 0; extra < PROCESS_MAX; extra++)
1860578         {
1860579           if (proc_table[extra].status == PROC_EMPTY   ||
1860580               proc_table[extra].status == PROC_ZOMBIE)
1860581             {
1860582               continue;
1860583             }
1860584           if (previous_segment_i == proc_table[extra].segment_i)
1860585             {
1860586               proc_count++;
1860587             }
1860588         }
1860589       if (proc_count == 0)
1860590         {
1860591           //
1860592           // The code segment can be released, because no other
1860593           // process is using it.
1860594           //
1860595           mb_free (previous_address_i, previous_size_i);
1860596         }
1860597     }
1860598   //
1860599   // Change the segment and the stack pointer, from the interrupt.
1860600   //
1860601   *segment_d = proc_table[pid].segment_d;
1860602   *sp        = proc_table[pid].sp;
1860603   //
1860604   return (0);
1860605 }
```

100.9.22 kernel/proc/proc_sys_exit.c

Si veda la sezione 99.8.21.

```
1870001 #include <kernel/proc.h>
1870002 #include <kernel/k_libc.h>
1870003 //----------------------------------------------------------------------
1870004 void
1870005 proc_sys_exit (pid_t pid, int status)
1870006 {
1870007   pid_t  child;
1870008   pid_t  parent = proc_table[pid].ppid;
```

```
1870009     pid_t   extra;
1870010     int     proc_count;
1870011     int     sigchld = 0;
1870012     int     fdn;
1870013     tty_t *tty;
1870014     //
1870015     proc_table[pid].status     = PROC_ZOMBIE;
1870016     proc_table[pid].ret        = status;
1870017     proc_table[pid].sig_status =  0;
1870018     proc_table[pid].sig_ignore =  0;
1870019     //
1870020     // Close files.
1870021     //
1870022     for (fdn = 0; fdn < OPEN_MAX; fdn++)
1870023       {
1870024         fd_close (pid, fdn);
1870025       }
1870026     //
1870027     // Close current directory.
1870028     //
1870029     inode_put (proc_table[pid].inode_cwd);
1870030     //
1870031     // Close the controlling terminal, if it is a process leader with
1870032     // such a terminal.
1870033     //
1870034     if (proc_table[pid].pgrp == pid && proc_table[pid].device_tty != 0)
1870035       {
1870036         tty = tty_reference (proc_table[pid].device_tty);
1870037         //
1870038         // Verify.
1870039         //
1870040         if (tty == NULL)
1870041           {
1870042             //
1870043             // Show a kernel message.
1870044             //
1870045             k_printf ("kernel alert: cannot find the terminal item "
1870046                       "for device 0x%04x!\n",
1870047                       (int) proc_table[pid].device_tty);
1870048           }
1870049         else if (tty->pgrp != pid)
1870050           {
1870051             //
1870052             // Show a kernel message.
1870053             //
1870054             k_printf ("kernel alert: terminal device 0x%04x should "
1870055                       "be associated to the process group %i, but it "
1870056                       "is instead related to process group %i!\n",
```

```
1870057                        (int) proc_table[pid].device_tty, (int) pid,
1870058                        (int) tty->pgrp);
1870059             }
1870060           else
1870061             {
1870062               tty->pgrp = 0;
1870063             }
1870064       }
1870065     //
1870066     // Free data segment memory.
1870067     //
1870068     mb_free (proc_table[pid].address_d, proc_table[pid].size_d);
1870069     //
1870070     // Free code segment memory if it is
1870071     // different from the data segment.
1870072     //
1870073     if (proc_table[pid].segment_i != proc_table[pid].segment_d)
1870074       {
1870075         //
1870076         // Must verify if no other process is using the same memory.
1870077         // The proc_count variable is incremented for processes
1870078         // active, ready or sleeping: the current process is already
1870079         // set as zombie, and is not counted.
1870080         //
1870081         for (proc_count = 0, extra = 0; extra < PROCESS_MAX; extra++)
1870082           {
1870083             if (proc_table[extra].status == PROC_EMPTY   ||
1870084                 proc_table[extra].status == PROC_ZOMBIE)
1870085               {
1870086                 continue;
1870087               }
1870088             if (proc_table[pid].segment_i == proc_table[extra].segment_i)
1870089               {
1870090                 proc_count++;
1870091               }
1870092           }
1870093         if (proc_count == 0)
1870094           {
1870095             //
1870096             // The code segment can be released, because no other
1870097             // process, except the current one (to be closed),
1870098             // is using it.
1870099             //
1870100             mb_free (proc_table[pid].address_i, proc_table[pid].size_i);
1870101           }
1870102       }
1870103     //
1870104     // Abandon children to 'init' ((pid_t) 1).
```

```
1870105     //
1870106     for (child = 1; child < PROCESS_MAX; child++)
1870107       {
1870108         if (   proc_table[child].status != PROC_EMPTY
1870109             && proc_table[child].ppid == pid)
1870110           {
1870111             proc_table[child].ppid = 1; // Son of `init'.
1870112             if (proc_table[child].status == PROC_ZOMBIE)
1870113               {
1870114                 sigchld = 1;    // Must send a SIGCHLD to `init'.
1870115               }
1870116           }
1870117       }
1870118     //
1870119     // SIGCHLD to `init'.
1870120     //
1870121     if (   sigchld
1870122         && pid != 1
1870123         && proc_table[1].status != PROC_EMPTY
1870124         && proc_table[1].status != PROC_ZOMBIE)
1870125       {
1870126         proc_sig_on ((pid_t) 1, SIGCHLD);
1870127       }
1870128     //
1870129     // Announce to the parent the death of its child.
1870130     //
1870131     if (   pid != parent
1870132         && proc_table[parent].status != PROC_EMPTY)
1870133       {
1870134         proc_sig_on (parent, SIGCHLD);
1870135       }
1870136 }
```

100.9.23 kernel/proc/proc_sys_fork.c

Si veda la sezione 99.8.22.

```
1880001 #include <kernel/proc.h>
1880002 #include <errno.h>
1880003 //----------------------------------------------------------------------
1880004 pid_t
1880005 proc_sys_fork (pid_t ppid, uint16_t sp)
1880006 {
1880007     pid_t     pid;
1880008     pid_t     zombie;
1880009     memory_t  allocated_i;
1880010     memory_t  allocated_d;
1880011     int       status;
```

```
1880012     int         fdn;
1880013     //
1880014     // Find a free PID.
1880015     //
1880016     for (pid = 1; pid < PROCESS_MAX; pid++)
1880017       {
1880018         if (proc_table[pid].status == PROC_EMPTY)
1880019           {
1880020             break;
1880021           }
1880022       }
1880023     if (pid >= PROCESS_MAX)
1880024       {
1880025         //
1880026         // There is no free pid.
1880027         //
1880028         errset (ENOMEM);         // Not enough space.
1880029         return (-1);
1880030       }
1880031     //
1880032     // Before allocating a new process, must check if there are some
1880033     // zombie slots, still with original segment data: should reset
1880034     // it now!
1880035     //
1880036     for (zombie = 1; zombie < PROCESS_MAX; zombie++)
1880037       {
1880038         if (   proc_table[zombie].status == PROC_ZOMBIE
1880039             && proc_table[zombie].segment_d != -1)
1880040           {
1880041             proc_table[zombie].segment_i = -1;     // Reset
1880042             proc_table[zombie].address_i = -1L;    // memory
1880043             proc_table[zombie].size_i    =  0;     // allocation
1880044             proc_table[zombie].segment_d = -1;     // data
1880045             proc_table[zombie].address_d = -1L;    // to
1880046             proc_table[zombie].size_d    =  0;     // impossible
1880047             proc_table[zombie].sp        =  0;     // values.
1880048           }
1880049       }
1880050     //
1880051     // Allocate memory: code and data segments.
1880052     //
1880053     if (proc_table[ppid].segment_i == proc_table[ppid].segment_d)
1880054       {
1880055         //
1880056         // Code segment and Data segment are the same
1880057         // (same I&D).
1880058         //
1880059         status = mb_alloc_size (proc_table[ppid].size_i, &allocated_i);
```

```
1880060             if (status < 0)
1880061               {
1880062                 errset (ENOMEM);          // Not enough space.
1880063                 return ((pid_t) -1);
1880064               }
1880065             allocated_d.address = allocated_i.address;
1880066             allocated_d.segment = allocated_i.segment;
1880067             allocated_d.size    = allocated_i.size;
1880068           }
1880069         else
1880070           {
1880071             //
1880072             // Code segment and Data segment are different
1880073             // (different I&D).
1880074             // Only the data segment is allocated.
1880075             //
1880076             status = mb_alloc_size (proc_table[ppid].size_d, &allocated_d);
1880077             if (status < 0)
1880078               {
1880079                 errset (ENOMEM);          // Not enough space.
1880080                 return ((pid_t) -1);
1880081               }
1880082             //
1880083             // Code segment is the same from the parent process.
1880084             //
1880085             allocated_i.address = proc_table[ppid].address_i;
1880086             allocated_i.segment = proc_table[ppid].segment_i;
1880087             allocated_i.size    = proc_table[ppid].size_i;
1880088           }
1880089         //
1880090         // Copy the process in memory.
1880091         //
1880092         if (proc_table[ppid].segment_i == proc_table[ppid].segment_d)
1880093           {
1880094             //
1880095             // Code segment and data segment are the same:
1880096             // must copy all.
1880097             //
1880098             // Copy the code segment: if the size is zero,
1880099             // it means 0x10000 bytes (65536 bytes).
1880100             //
1880101             if (proc_table[ppid].size_i == 0)
1880102               {
1880103                 //
1880104                 // Copy 0x10000 bytes with two steps.
1880105                 //
1880106                 mem_copy (proc_table[ppid].address_i,
1880107                           allocated_i.address, 0x8000);
```

```
1880108                 mem_copy ((proc_table[ppid].address_i + 0x8000),
1880109                           (allocated_i.address + 0x8000), 0x8000);
1880110               }
1880111             else
1880112               {
1880113                 //
1880114                 // Normal copy.
1880115                 //
1880116                 mem_copy (proc_table[ppid].address_i, allocated_i.address,
1880117                           proc_table[ppid].size_i);
1880118               }
1880119           }
1880120         else
1880121           {
1880122             //
1880123             // Code segment and data segment are different:
1880124             // copy only the data segment.
1880125             //
1880126             // Copy the data segment in memory: if the size is zero,
1880127             // it means 0x10000 bytes (65536 bytes).
1880128             //
1880129             if (proc_table[ppid].size_d == 0)
1880130               {
1880131                 //
1880132                 // Copy 0x10000 bytes with two steps.
1880133                 //
1880134                 mem_copy (proc_table[ppid].address_d,
1880135                           allocated_d.address, 0x8000);
1880136                 mem_copy ((proc_table[ppid].address_d + 0x8000),
1880137                           (allocated_d.address + 0x8000), 0x8000);
1880138               }
1880139             else
1880140               {
1880141                 //
1880142                 // Normal copy.
1880143                 //
1880144                 mem_copy (proc_table[ppid].address_d, allocated_d.address,
1880145                           proc_table[ppid].size_d);
1880146               }
1880147           }
1880148         //
1880149         // Allocate the new PID.
1880150         //
1880151         proc_table[pid].ppid          = ppid;
1880152         proc_table[pid].pgrp          = proc_table[ppid].pgrp;
1880153         proc_table[pid].uid           = proc_table[ppid].uid;
1880154         proc_table[pid].euid          = proc_table[ppid].euid;
1880155         proc_table[pid].suid          = proc_table[ppid].suid;
```

```
1880156      proc_table[pid].device_tty    = proc_table[ppid].device_tty;
1880157      proc_table[pid].sig_status    = 0;
1880158      proc_table[pid].sig_ignore    = 0;
1880159      proc_table[pid].usage         = 0;
1880160      proc_table[pid].status        = PROC_CREATED;
1880161      proc_table[pid].wakeup_events = 0;
1880162      proc_table[pid].wakeup_signal = 0;
1880163      proc_table[pid].wakeup_timer  = 0;
1880164      proc_table[pid].segment_i     = allocated_i.segment;
1880165      proc_table[pid].address_i     = allocated_i.address;
1880166      proc_table[pid].size_i        = proc_table[ppid].size_i;
1880167      proc_table[pid].segment_d     = allocated_d.segment;
1880168      proc_table[pid].address_d     = allocated_d.address;
1880169      proc_table[pid].size_d        = proc_table[ppid].size_d;
1880170      proc_table[pid].sp            = sp;
1880171      proc_table[pid].ret           = 0;
1880172      proc_table[pid].inode_cwd     = proc_table[ppid].inode_cwd;
1880173      proc_table[pid].umask         = proc_table[ppid].umask;
1880174      strncpy (proc_table[pid].name,
1880175               proc_table[ppid].name, PATH_MAX);
1880176      strncpy (proc_table[pid].path_cwd,
1880177               proc_table[ppid].path_cwd, PATH_MAX);
1880178      //
1880179      // Increase inode references for the working directory.
1880180      //
1880181      proc_table[pid].inode_cwd->references++;
1880182      //
1880183      // Duplicate valid file descriptors.
1880184      //
1880185      for (fdn = 0; fdn < OPEN_MAX; fdn++)
1880186        {
1880187          if (   proc_table[ppid].fd[fdn].file != NULL
1880188              && proc_table[ppid].fd[fdn].file->inode != NULL)
1880189            {
1880190              //
1880191              // Copy to the forked process.
1880192              //
1880193              proc_table[pid].fd[fdn].fl_flags
1880194                  = proc_table[ppid].fd[fdn].fl_flags;
1880195              proc_table[pid].fd[fdn].fd_flags
1880196                  = proc_table[ppid].fd[fdn].fd_flags;
1880197              proc_table[pid].fd[fdn].file
1880198                  = proc_table[ppid].fd[fdn].file;
1880199              //
1880200              // Increment file reference.
1880201              //
1880202              proc_table[ppid].fd[fdn].file->references++;
1880203            }
```

```
1880204         }
1880205     //
1880206     // Change segment values inside the stack: DS==ES; CS.
1880207     //
1880208     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1880209             (allocated_d.address + proc_table[pid].sp + 14),
1880210             &allocated_d.segment, (sizeof allocated_d.segment), NULL);
1880211     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1880212             (allocated_d.address + proc_table[pid].sp + 16),
1880213             &allocated_d.segment, (sizeof allocated_d.segment), NULL);
1880214     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1880215             (allocated_d.address + proc_table[pid].sp + 20),
1880216             &allocated_i.segment, (sizeof allocated_i.segment), NULL);
1880217     //
1880218     // Set it ready.
1880219     //
1880220     proc_table[pid].status  = PROC_READY;
1880221     //
1880222     // Return the new PID.
1880223     //
1880224     return (pid);
1880225 }
```

100.9.24 kernel/proc/proc_sys_kill.c

«

Si veda la sezione 99.8.23.

```
1890001 #include <kernel/proc.h>
1890002 #include <errno.h>
1890003 //----------------------------------------------------------------------
1890004 int
1890005 proc_sys_kill (pid_t pid_killer, pid_t pid_target, int sig)
1890006 {
1890007     uid_t euid = proc_table[pid_killer].euid;
1890008     uid_t uid  = proc_table[pid_killer].uid;
1890009     pid_t pgrp = proc_table[pid_killer].pgrp;
1890010     int   p;                      // Index inside the process table.
1890011     //
1890012     if (pid_target < -1)
1890013       {
1890014         errset (ESRCH);
1890015         return (-1);
1890016       }
1890017     else if (pid_target == -1)
1890018       {
1890019         if (sig == 0)
1890020           {
1890021             return (0);
```

```
1890022             }
1890023           if (euid == 0)
1890024             {
1890025               //
1890026               // Because 'pid_target' is qual to '-1' and the effective
1890027               // user identity is '0', then, all processes,
1890028               // except the kernel and init, will receive the signal.
1890029               //
1890030               // The following scan starts from 2, to preserve the
1890031               // kernel and init processes.
1890032               //
1890033               for (p = 2; p < PROCESS_MAX; p++)
1890034                 {
1890035                   if (   proc_table[p].status != PROC_EMPTY
1890036                       && proc_table[p].status != PROC_ZOMBIE)
1890037                     {
1890038                       proc_sig_on (p, sig);
1890039                     }
1890040                 }
1890041             }
1890042           else
1890043             {
1890044               //
1890045               // Because 'pid_target' is qual to '-1', but the effective
1890046               // user identity is not '0', then, all processes owned
1890047               // by the same effective user identity, will receive the
1890048               // signal.
1890049               //
1890050               // The following scan starts from 1, to preserve the
1890051               // kernel process.
1890052               //
1890053               for (p = 1; p < PROCESS_MAX; p++)
1890054                 {
1890055                   if (   proc_table[p].status != PROC_EMPTY
1890056                       && proc_table[p].status != PROC_ZOMBIE
1890057                       && proc_table[p].uid == euid)
1890058                     {
1890059                       proc_sig_on (p, sig);
1890060                     }
1890061                 }
1890062             }
1890063           return (0);
1890064         }
1890065       else if (pid_target == 0)
1890066         {
1890067           if (sig == 0)
1890068             {
1890069               return (0);
```

```
1890070             }
1890071           //
1890072           // The following scan starts from 1, to preserve the
1890073           // kernel process.
1890074           //
1890075           for (p = 1; p < PROCESS_MAX; p++)
1890076             {
1890077               if (   proc_table[p].status != PROC_EMPTY
1890078                   && proc_table[p].status != PROC_ZOMBIE
1890079                   && proc_table[p].pgrp == pgrp)
1890080                 {
1890081                   proc_sig_on (p, sig);
1890082                 }
1890083             }
1890084           return (0);
1890085         }
1890086       else if (pid_target >= PROCESS_MAX)
1890087         {
1890088           errset (ESRCH);
1890089           return (-1);
1890090         }
1890091       else // (pid_target > 0)
1890092         {
1890093           if (proc_table[pid_target].status == PROC_EMPTY  ||
1890094               proc_table[pid_target].status == PROC_ZOMBIE)
1890095             {
1890096               errset (ESRCH);
1890097               return (-1);
1890098             }
1890099           else if (uid  == proc_table[pid_target].uid   ||
1890100                    uid  == proc_table[pid_target].suid  ||
1890101                    euid == proc_table[pid_target].uid   ||
1890102                    euid == proc_table[pid_target].suid  ||
1890103                    euid == 0)
1890104             {
1890105               if (sig == 0)
1890106                 {
1890107                   return (0);
1890108                 }
1890109               else
1890110                 {
1890111                   proc_sig_on (pid_target, sig);
1890112                   return (0);
1890113                 }
1890114             }
1890115           else
1890116             {
1890117               errset (EPERM);
```

```
1890118                return (-1);
1890119              }
1890120          }
1890121  }
```

100.9.25 kernel/proc/proc_sys_seteuid.c

«

Si veda la sezione 99.8.24.

```
1900001  #include <kernel/proc.h>
1900002  #include <errno.h>
1900003  //----------------------------------------------------------------------
1900004  int
1900005  proc_sys_seteuid (pid_t pid, uid_t euid)
1900006  {
1900007      if (proc_table[pid].euid == 0)
1900008        {
1900009          proc_table[pid].euid = euid;
1900010          return (0);
1900011        }
1900012      else if (euid == proc_table[pid].euid)
1900013        {
1900014          return (0);
1900015        }
1900016      else if (euid == proc_table[pid].uid || euid == proc_table[pid].suid)
1900017        {
1900018          proc_table[pid].euid = euid;
1900019          return (0);
1900020        }
1900021      else
1900022        {
1900023          errset (EPERM);
1900024          return (-1);
1900025        }
1900026  }
```

100.9.26 kernel/proc/proc_sys_setuid.c

«

Si veda la sezione 99.8.25.

```
1910001  #include <kernel/proc.h>
1910002  #include <errno.h>
1910003  //----------------------------------------------------------------------
1910004  int
1910005  proc_sys_setuid (pid_t pid, uid_t uid)
1910006  {
1910007      if (proc_table[pid].euid == 0)
```

```
1910008         {
1910009           proc_table[pid].uid  = uid;
1910010           proc_table[pid].euid = uid;
1910011           proc_table[pid].suid = uid;
1910012           return (0);
1910013         }
1910014       else if (uid == proc_table[pid].euid)
1910015         {
1910016           return (0);
1910017         }
1910018       else if (uid == proc_table[pid].uid || uid == proc_table[pid].suid)
1910019         {
1910020           proc_table[pid].euid = uid;
1910021           return (0);
1910022         }
1910023       else
1910024         {
1910025           errset (EPERM);
1910026           return (-1);
1910027         }
1910028 }
```

100.9.27 kernel/proc/proc_sys_signal.c

«

Si veda la sezione 99.8.26.

```
1920001 #include <kernel/proc.h>
1920002 #include <errno.h>
1920003 //----------------------------------------------------------------------
1920004 sighandler_t
1920005 proc_sys_signal (pid_t pid, int sig, sighandler_t handler)
1920006 {
1920007     unsigned long int flag = 1L << (sig - 1);
1920008     sighandler_t      previous;
1920009
1920010     if (sig <= 0)
1920011       {
1920012         errset (EINVAL);
1920013         return (SIG_ERR);
1920014       }
1920015
1920016     if (proc_table[pid].sig_ignore & flag)
1920017       {
1920018         previous = SIG_IGN;
1920019       }
1920020     else
1920021       {
1920022         previous = SIG_DFL;
```

```
1920023         }
1920024
1920025     if (handler == SIG_DFL)
1920026       {
1920027         proc_table[pid].sig_ignore ^= flag;
1920028         return (previous);
1920029       }
1920030     else if (handler == SIG_IGN)
1920031       {
1920032         proc_table[pid].sig_ignore |= flag;
1920033         return (previous);
1920034       }
1920035     else
1920036       {
1920037         errset (EINVAL);
1920038         return (SIG_ERR);
1920039       }
1920040 }
```

100.9.28 kernel/proc/proc_sys_wait.c

«

Si veda la sezione 99.8.27.

```
1930001 #include <kernel/proc.h>
1930002 #include <errno.h>
1930003 //----------------------------------------------------------------------
1930004 pid_t
1930005 proc_sys_wait (pid_t pid, int *status)
1930006 {
1930007     pid_t  parent = pid;
1930008     pid_t  child;
1930009     int    child_available = 0;
1930010     //
1930011     // Find a dead child process.
1930012     //
1930013     for (child = 1; child < PROCESS_MAX; child++)
1930014       {
1930015         if (proc_table[child].ppid == parent)
1930016           {
1930017             child_available = 1;        // Child found!
1930018             if (proc_table[child].status == PROC_ZOMBIE)
1930019               {
1930020                 break;                  // It is dead!
1930021               }
1930022           }
1930023       }
1930024     //
1930025     // If the index 'child' is a valid process number,
```

```
1930026         // a dead child was found.
1930027         //
1930028         if (child < PROCESS_MAX)
1930029           {
1930030             *status = proc_table[child].ret;
1930031             proc_available (child);
1930032             return (child);
1930033           }
1930034         else
1930035           {
1930036             if (child_available)
1930037               {
1930038                 //
1930039                 // There are child, but all alive.
1930040                 //
1930041                 // Go to sleep.
1930042                 //
1930043                 proc_table[parent].status          = PROC_SLEEPING;
1930044                 proc_table[parent].wakeup_events |= WAKEUP_EVENT_SIGNAL;
1930045                 proc_table[parent].wakeup_signal  = SIGCHLD;
1930046                 return ((pid_t) 0);
1930047               }
1930048             else
1930049               {
1930050                 //
1930051                 // There are no child at all.
1930052                 //
1930053                 errset (ECHILD);
1930054                 return ((pid_t) -1);
1930055               }
1930056           }
1930057 }
```

100.9.29 kernel/proc/proc_table.c

«

Si veda la sezione 99.8.7.

```
1940001 #include <kernel/proc.h>
1940002 //----------------------------------------------------------------------
1940003 proc_t   proc_table[PROCESS_MAX];
```

100.9.30 kernel/proc/sysroutine.c

Si veda la sezione 99.8.28.

```
1950001 #include <kernel/proc.h>
1950002 #include <errno.h>
1950003 #include <kernel/k_libc.h>
1950004 //----------------------------------------------------------------------
1950005 static void sysroutine_error_back (int *number, int *line,
1950006                                    char *file_name);
1950007 //----------------------------------------------------------------------
1950008 void
1950009 sysroutine (uint16_t *sp, segment_t *segment_d, uint16_t syscallnr,
1950010             uint16_t msg_off, uint16_t msg_size)
1950011 {
1950012   pid_t  pid      = proc_find (*segment_d);
1950013   addr_t msg_addr = address (*segment_d, msg_off);
1950014   //
1950015   // Inbox.
1950016   //
1950017   union {
1950018       sysmsg_chdir_t     chdir;
1950019       sysmsg_chmod_t     chmod;
1950020       sysmsg_chown_t     chown;
1950021       sysmsg_clock_t     clock;
1950022       sysmsg_close_t     close;
1950023       sysmsg_dup_t       dup;
1950024       sysmsg_dup2_t      dup2;
1950025       sysmsg_exec_t      exec;
1950026       sysmsg_exit_t      exit;
1950027       sysmsg_fchmod_t    fchmod;
1950028       sysmsg_fchown_t    fchown;
1950029       sysmsg_fcntl_t     fcntl;
1950030       sysmsg_fork_t      fork;
1950031       sysmsg_fstat_t     fstat;
1950032       sysmsg_kill_t      kill;
1950033       sysmsg_link_t      link;
1950034       sysmsg_lseek_t     lseek;
1950035       sysmsg_mkdir_t     mkdir;
1950036       sysmsg_mknod_t     mknod;
1950037       sysmsg_mount_t     mount;
1950038       sysmsg_open_t      open;
1950039       sysmsg_read_t      read;
1950040       sysmsg_seteuid_t   seteuid;
1950041       sysmsg_setuid_t    setuid;
1950042       sysmsg_signal_t    signal;
1950043       sysmsg_sleep_t     sleep;
1950044       sysmsg_stat_t      stat;
1950045       sysmsg_stime_t     stime;
```

```
1950046         sysmsg_time_t      time;
1950047         sysmsg_uarea_t     uarea;
1950048         sysmsg_umask_t     umask;
1950049         sysmsg_umount_t    umount;
1950050         sysmsg_unlink_t    unlink;
1950051         sysmsg_wait_t      wait;
1950052         sysmsg_write_t     write;
1950053         sysmsg_zpchar_t    zpchar;
1950054         sysmsg_zpstring_t  zpstring;
1950055     } msg;
1950056     //
1950057     // Verify if the system call was emitted from kernel code.
1950058     // The kernel can emit only some particular system call:
1950059     //   SYS_NULL, to let other processes run;
1950060     //   SYS_FORK, to let fork itself;
1950061     //   SYS_EXEC, to replace a forked copy of itself.
1950062     //
1950063     if (pid == 0)
1950064       {
1950065         //
1950066         // This is the kernel code!
1950067         //
1950068         if (   syscallnr != SYS_0
1950069             && syscallnr != SYS_FORK
1950070             && syscallnr != SYS_EXEC)
1950071           {
1950072             k_printf ("kernel panic: the system call %i ", syscallnr);
1950073             k_printf ("was received while running in kernel space!\n");
1950074           }
1950075       }
1950076     //
1950077     // Entering a system call, the kernel variable 'errno' must be
1950078     // reset, otherwise, a previous kernel code error might be returned
1950079     // to the applications.
1950080     //
1950081     errno    = 0;
1950082     errln    = 0;
1950083     errfn[0] = 0;
1950084     //
1950085     // Get message.
1950086     //
1950087     dev_io (pid, DEV_MEM, DEV_READ, msg_addr, &msg, msg_size, NULL);
1950088     //
1950089     // Do the request from the received system call.
1950090     //
1950091     switch (syscallnr)
1950092       {
1950093         case SYS_0:
```

```
1950094             break;
1950095         case SYS_CHDIR:
1950096             msg.chdir.ret          = path_chdir (pid, msg.chdir.path);
1950097             sysroutine_error_back (&msg.chdir.errno, &msg.chdir.errln,
1950098                                    msg.chdir.errfn);
1950099             break;
1950100         case SYS_CHMOD:
1950101             msg.chmod.ret          = path_chmod (pid, msg.chmod.path,
1950102                                                  msg.chmod.mode);
1950103             sysroutine_error_back (&msg.chmod.errno, &msg.chmod.errln,
1950104                                    msg.chmod.errfn);
1950105             break;
1950106         case SYS_CHOWN:
1950107             msg.chown.ret          = path_chown (pid, msg.chown.path,
1950108                                                  msg.chown.uid,
1950109                                                  msg.chown.gid);
1950110             sysroutine_error_back (&msg.chown.errno, &msg.chown.errln,
1950111                                    msg.chown.errfn);
1950112             break;
1950113         case SYS_CLOCK:
1950114             msg.clock.ret          = k_clock ();
1950115             break;
1950116         case SYS_CLOSE:
1950117             msg.close.ret          = fd_close (pid, msg.close.fdn);
1950118             sysroutine_error_back (&msg.close.errno, &msg.close.errln,
1950119                                    msg.close.errfn);
1950120             break;
1950121         case SYS_DUP:
1950122             msg.dup.ret            = fd_dup (pid, msg.dup.fdn_old, 0);
1950123             sysroutine_error_back (&msg.dup.errno, &msg.dup.errln,
1950124                                    msg.dup.errfn);
1950125             break;
1950126         case SYS_DUP2:
1950127             msg.dup2.ret           = fd_dup2 (pid, msg.dup2.fdn_old,
1950128                                               msg.dup2.fdn_new);
1950129             sysroutine_error_back (&msg.dup2.errno, &msg.dup2.errln,
1950130                                    msg.dup2.errfn);
1950131             break;
1950132         case SYS_EXEC:
1950133             msg.exec.ret           = proc_sys_exec (sp, segment_d, pid,
1950134                                                     msg.exec.path,
1950135                                                     msg.exec.argc,
1950136                                                     msg.exec.arg_data,
1950137                                                     msg.exec.envc,
1950138                                                     msg.exec.env_data);
1950139             msg.exec.uid           = proc_table[pid].uid;
1950140             msg.exec.euid          = proc_table[pid].euid;
1950141             sysroutine_error_back (&msg.exec.errno, &msg.exec.errln,
```

```
1950142                                  msg.exec.errfn);
1950143             break;
1950144         case SYS_EXIT:
1950145             if (pid == 0)
1950146               {
1950147                 k_printf ("kernel alert: "
1950148                           "the kernel cannot exit!\n");
1950149               }
1950150             else
1950151               {
1950152                 proc_sys_exit (pid, msg.exit.status);
1950153               }
1950154             break;
1950155         case SYS_FCHMOD:
1950156             msg.fchmod.ret        = fd_chmod (pid, msg.fchmod.fdn,
1950157                                               msg.fchmod.mode);
1950158             sysroutine_error_back (&msg.fchmod.errno, &msg.fchmod.errln,
1950159                                    msg.fchmod.errfn);
1950160             break;
1950161         case SYS_FCHOWN:
1950162             msg.fchown.ret        = fd_chown (pid, msg.fchown.fdn,
1950163                                               msg.fchown.uid,
1950164                                               msg.fchown.gid);
1950165             sysroutine_error_back (&msg.fchown.errno, &msg.fchown.errln,
1950166                                    msg.fchown.errfn);
1950167             break;
1950168         case SYS_FCNTL:
1950169             msg.fcntl.ret         = fd_fcntl (pid, msg.fcntl.fdn,
1950170                                               msg.fcntl.cmd,
1950171                                               msg.fcntl.arg);
1950172             sysroutine_error_back (&msg.fcntl.errno, &msg.fcntl.errln,
1950173                                    msg.fcntl.errfn);
1950174             break;
1950175         case SYS_FORK:
1950176             msg.fork.ret          = proc_sys_fork (pid, *sp);
1950177             sysroutine_error_back (&msg.fork.errno, &msg.fork.errln,
1950178                                    msg.fork.errfn);
1950179             break;
1950180         case SYS_FSTAT:
1950181             msg.fstat.ret         = fd_stat (pid, msg.fstat.fdn,
1950182                                              &msg.fstat.stat);
1950183             sysroutine_error_back (&msg.fstat.errno, &msg.fstat.errln,
1950184                                    msg.fstat.errfn);
1950185             break;
1950186         case SYS_KILL:
1950187             msg.kill.ret          = proc_sys_kill (pid, msg.kill.pid,
1950188                                                    msg.kill.signal);
1950189             sysroutine_error_back (&msg.kill.errno, &msg.kill.errln,
```

```
1950190                                 msg.kill.errfn);
1950191             break;
1950192         case SYS_LINK:
1950193             msg.link.ret          = path_link (pid, msg.link.path_old,
1950194                                                 msg.link.path_new);
1950195             sysroutine_error_back (&msg.link.errno, &msg.link.errln,
1950196                                    msg.link.errfn);
1950197             break;
1950198         case SYS_LSEEK:
1950199             msg.lseek.ret         = fd_lseek (pid, msg.lseek.fdn,
1950200                                                msg.lseek.offset,
1950201                                                msg.lseek.whence);
1950202             sysroutine_error_back (&msg.lseek.errno, &msg.lseek.errln,
1950203                                    msg.lseek.errfn);
1950204             break;
1950205         case SYS_MKDIR:
1950206             msg.mkdir.ret         = path_mkdir (pid, msg.mkdir.path,
1950207                                                  msg.mkdir.mode);
1950208             sysroutine_error_back (&msg.mkdir.errno, &msg.mkdir.errln,
1950209                                    msg.mkdir.errfn);
1950210             break;
1950211         case SYS_MKNOD:
1950212             msg.mknod.ret         = path_mknod (pid, msg.mknod.path,
1950213                                                  msg.mknod.mode,
1950214                                                  msg.mknod.device);
1950215             sysroutine_error_back (&msg.mknod.errno, &msg.mknod.errln,
1950216                                    msg.mknod.errfn);
1950217             break;
1950218         case SYS_MOUNT:
1950219             msg.mount.ret         = path_mount (pid, msg.mount.path_dev,
1950220                                                  msg.mount.path_mnt,
1950221                                                  msg.mount.options);
1950222             sysroutine_error_back (&msg.mount.errno, &msg.mount.errln,
1950223                                    msg.mount.errfn);
1950224             break;
1950225         case SYS_OPEN:
1950226             msg.open.ret          = fd_open (pid, msg.open.path,
1950227                                               msg.open.flags,
1950228                                               msg.open.mode);
1950229             sysroutine_error_back (&msg.open.errno, &msg.open.errln,
1950230                                    msg.open.errfn);
1950231             break;
1950232         case SYS_PGRP:
1950233             proc_table[pid].pgrp          = pid;
1950234             break;
1950235         case SYS_READ:
1950236             msg.read.ret          = fd_read (pid, msg.read.fdn,
1950237                                               msg.read.buffer,
```

```
1950238                                               msg.read.count,
1950239                                               &msg.read.eof);
1950240             sysroutine_error_back (&msg.read.errno, &msg.read.errln,
1950241                                    msg.read.errfn);
1950242             break;
1950243         case SYS_SETEUID:
1950244             msg.seteuid.ret       = proc_sys_seteuid (pid,
1950245                                                        msg.seteuid.euid);
1950246             msg.seteuid.euid      = proc_table[pid].euid;
1950247             sysroutine_error_back (&msg.seteuid.errno, &msg.seteuid.errln,
1950248                                    msg.seteuid.errfn);
1950249             break;
1950250         case SYS_SETUID:
1950251             msg.setuid.ret        = proc_sys_setuid (pid,
1950252                                                       msg.setuid.euid);
1950253             msg.setuid.uid        = proc_table[pid].uid;
1950254             msg.setuid.euid       = proc_table[pid].euid;
1950255             msg.setuid.suid       = proc_table[pid].suid;
1950256             sysroutine_error_back (&msg.setuid.errno, &msg.setuid.errln,
1950257                                    msg.setuid.errfn);
1950258             break;
1950259         case SYS_SIGNAL:
1950260             msg.signal.ret        = proc_sys_signal (pid,
1950261                                                       msg.signal.signal,
1950262                                                      msg.signal.handler);
1950263             sysroutine_error_back (&msg.signal.errno, &msg.signal.errln,
1950264                                    msg.signal.errfn);
1950265             break;
1950266         case SYS_SLEEP:
1950267             proc_table[pid].status        = PROC_SLEEPING;
1950268             proc_table[pid].ret           = 0;
1950269             proc_table[pid].wakeup_events = msg.sleep.events;
1950270             proc_table[pid].wakeup_signal = msg.sleep.signal;
1950271             proc_table[pid].wakeup_timer  = msg.sleep.seconds;
1950272             break;
1950273         case SYS_STAT:
1950274             msg.stat.ret          = path_stat (pid, msg.stat.path,
1950275                                                 &msg.stat.stat);
1950276             sysroutine_error_back (&msg.stat.errno, &msg.stat.errln,
1950277                                    msg.stat.errfn);
1950278             break;
1950279         case SYS_STIME:
1950280             msg.stime.ret         = k_stime (&msg.stime.timer);
1950281             break;
1950282         case SYS_TIME:
1950283             msg.time.ret          = k_time (NULL);
1950284             break;
1950285         case SYS_UAREA:
```

```
1950286             msg.uarea.uid          = proc_table[pid].uid;
1950287             msg.uarea.euid         = proc_table[pid].euid;
1950288             msg.uarea.pid          = pid;
1950289             msg.uarea.ppid         = proc_table[pid].ppid;
1950290             msg.uarea.pgrp         = proc_table[pid].pgrp;
1950291             msg.uarea.umask        = proc_table[pid].umask;
1950292             strncpy (msg.uarea.path_cwd,
1950293                      proc_table[pid].path_cwd, PATH_MAX);
1950294             break;
1950295         case SYS_UMASK:
1950296             msg.umask.ret          = proc_table[pid].umask;
1950297             proc_table[pid].umask = (msg.umask.umask & 00777);
1950298             break;
1950299         case SYS_UMOUNT:
1950300             msg.umount.ret         = path_umount (pid,
1950301                                                 msg.umount.path_mnt);
1950302             sysroutine_error_back (&msg.umount.errno, &msg.umount.errln,
1950303                                    msg.umount.errfn);
1950304             break;
1950305         case SYS_UNLINK:
1950306             msg.unlink.ret         = path_unlink (pid, msg.unlink.path);
1950307             sysroutine_error_back (&msg.unlink.errno, &msg.unlink.errln,
1950308                                    msg.unlink.errfn);
1950309             break;
1950310         case SYS_WAIT:
1950311             msg.wait.ret           = proc_sys_wait (pid,
1950312                                                   &msg.wait.status);
1950313             sysroutine_error_back (&msg.wait.errno, &msg.wait.errln,
1950314                                    msg.wait.errfn);
1950315             break;
1950316         case SYS_WRITE:
1950317             msg.write.ret          = fd_write (pid, msg.write.fdn,
1950318                                              msg.write.buffer,
1950319                                              msg.write.count);
1950320             sysroutine_error_back (&msg.write.errno, &msg.write.errln,
1950321                                    msg.write.errfn);
1950322             break;
1950323         case SYS_ZPCHAR:
1950324             dev_io (pid, DEV_TTY, DEV_WRITE, 0L, &msg.zpchar.c,
1950325                     1, NULL);
1950326             break;
1950327         case SYS_ZPSTRING:
1950328             dev_io (pid, DEV_TTY, DEV_WRITE, 0L,
1950329                     msg.zpstring.string,
1950330                     strlen (msg.zpstring.string), NULL);
1950331             break;
1950332         default:
1950333             k_printf ("kernel alert: unknown system call %i!\n",
```

```
1950334                         syscallnr);
1950335               break;
1950336         }
1950337     //
1950338     // Return value with a message back.
1950339     //
1950340     dev_io (pid, DEV_MEM, DEV_WRITE, msg_addr, &msg, msg_size, NULL);
1950341     //
1950342     // Continue with the scheduler.
1950343     //
1950344     proc_scheduler (sp, segment_d);
1950345 }
1950346 //----------------------------------------------------------------------
1950347 static void
1950348 sysroutine_error_back (int *number, int *line, char *file_name)
1950349 {
1950350     *number = errno;
1950351     *line   = errln;
1950352     strncpy (file_name, errfn, PATH_MAX);
1950353     file_name[PATH_MAX-1] = 0;
1950354 }
```

100.10 os16: «kernel/tty.h»

Si veda la sezione 99.9.

```
1960001 #ifndef _KERNEL_TTY_H
1960002 #define _KERNEL_TTY_H  1
1960003 
1960004 #include <stddef.h>
1960005 #include <stdint.h>
1960006 #include <sys/types.h>
1960007 #include <kernel/ibm_i86.h>
1960008 
1960009 //----------------------------------------------------------------------
1960010 #define         TTYS_CONSOLE    IBM_I86_VIDEO_PAGES
1960011 #define         TTYS_SERIAL     0
1960012 #define         TTYS_TOTAL      (TTYS_CONSOLE + TTYS_SERIAL)
1960013 //----------------------------------------------------------------------
1960014 #define         TTY_OK          0
1960015 #define         TTY_LOST_KEY    1
1960016 //----------------------------------------------------------------------
1960017 typedef struct {
1960018     dev_t       device;
1960019     pid_t       pgrp;           // Process group.
1960020     int         key;            // Last pressed key.
1960021     int         status;         // 0 = ok, 1 = lost typed character.
1960022 } tty_t;
```

```
1960023 //----------------------------------------------------------------------
1960024 extern tty_t tty_table[TTYS_TOTAL];
1960025 //----------------------------------------------------------------------
1960026 tty_t *tty_reference (dev_t device);
1960027 dev_t tty_console   (dev_t device);
1960028 int    tty_read      (dev_t device);
1960029 void   tty_write     (dev_t device, int c);
1960030 void   tty_init      (void);
1960031
1960032 #endif
```

100.10.1 kernel/tty/tty_console.c

Si veda la sezione 99.9.

```
1970001 #include <sys/os16.h>
1970002 #include <kernel/tty.h>
1970003 //----------------------------------------------------------------------
1970004 dev_t
1970005 tty_console (dev_t device)
1970006 {
1970007     static dev_t device_active = DEV_CONSOLE0;  // First time.
1970008            dev_t device_previous;
1970009     //
1970010     // Check if it required only the current device.
1970011     //
1970012     if (device == 0)
1970013       {
1970014         return (device_active);
1970015       }
1970016     //
1970017     // Fix if the device is not valid.
1970018     //
1970019     if (device > DEV_CONSOLE3 || device < DEV_CONSOLE0)
1970020       {
1970021         device = DEV_CONSOLE0;
1970022       }
1970023     //
1970024     // Update.
1970025     //
1970026     device_previous = device_active;
1970027     device_active   = device;
1970028     //
1970029     // Switch.
1970030     //
1970031     con_select (device_active & 0x00FF);
1970032     //
1970033     // Return previous device value.
```

```
1970034     //
1970035     return (device_previous);
1970036 }
```

100.10.2 kernel/tty/tty_init.c

«

Si veda la sezione 99.9.

```
1980001 #include <sys/os16.h>
1980002 #include <kernel/tty.h>
1980003 //----------------------------------------------------------------------
1980004 void
1980005 tty_init (void)
1980006 {
1980007     int page;   // console page.
1980008     //
1980009     // Console initialization: console pages correspond to the first
1980010     // terminal items.
1980011     //
1980012     for (page = 0; page < TTYS_CONSOLE; page++)
1980013       {
1980014         tty_table[page].device = DEV_CONSOLE0 + page;
1980015         tty_table[page].pgrp   = 0;
1980016         tty_table[page].key    = 0;
1980017         tty_table[page].status = TTY_OK;
1980018       }
1980019     //
1980020     // Set video mode.
1980021     //
1980022     con_init ();
1980023     //
1980024     // Select the first console.
1980025     //
1980026     tty_console (DEV_CONSOLE0);
1980027     //
1980028     // Nothing else to configure (only consoles are available).
1980029     //
1980030     return;
1980031 }
```

100.10.3 kernel/tty/tty_read.c

«

Si veda la sezione 99.9.

```
1990001 #include <sys/os16.h>
1990002 #include <kernel/tty.h>
1990003 #include <kernel/k_libc.h>
```

```
1990004 //----------------------------------------------------------------------
1990005 int
1990006 tty_read (dev_t device)
1990007 {
1990008     tty_t   *tty;
1990009     int      key;
1990010     //
1990011     tty = tty_reference (device);
1990012     if (tty == NULL)
1990013       {
1990014         k_printf ("kernel alert: cannot find terminal device "
1990015                   "0x%04x!\n", (int) device);
1990016         //
1990017         return (0);
1990018       }
1990019     //
1990020     // Read key and remove from the source.
1990021     //
1990022     key      = tty->key;
1990023     tty->key = 0;
1990024     //
1990025     // Return the key.
1990026     //
1990027     return (key);
1990028
1990029 }
```

100.10.4 kernel/tty/tty_reference.c

Si veda la sezione 99.9.

```
2000001 #include <kernel/tty.h>
2000002 //----------------------------------------------------------------------
2000003 tty_t *
2000004 tty_reference (dev_t device)
2000005 {
2000006     int t;              // Terminal index.
2000007     //
2000008     // If device is zero, a reference to the whole table is returned.
2000009     //
2000010     if (device == 0)
2000011       {
2000012         return (tty_table);
2000013       }
2000014     //
2000015     // Otherwise, a scan is made to find the selected device.
2000016     //
2000017     for (t = 0; t < TTYS_TOTAL; t++)
```

```
2000018         {
2000019           if (tty_table[t].device == device)
2000020             {
2000021               //
2000022               // Device found. Return the pointer.
2000023               //
2000024               return (&tty_table[t]);
2000025             }
2000026         }
2000027     //
2000028     // No device found!
2000029     //
2000030     return (NULL);
2000031 }
```

100.10.5 kernel/tty/tty_table.c

«

Si veda la sezione 99.9.

```
2010001 #include <kernel/tty.h>
2010002 //----------------------------------------------------------------------
2010003 tty_t tty_table[TTYS_TOTAL];
```

100.10.6 kernel/tty/tty_write.c

«

Si veda la sezione 99.9.

```
2020001 #include <sys/os16.h>
2020002 #include <kernel/tty.h>
2020003 //----------------------------------------------------------------------
2020004 void
2020005 tty_write (dev_t device, int c)
2020006 {
2020007     int   console;
2020008     //
2020009     if ((device & 0xFF00) == (DEV_CONSOLE_MAJOR << 8))
2020010       {
2020011         console = (device & 0x00FF);
2020012         con_putc (console, c);
2020013       }
2020014 }
```

Capitolo 101

Sorgenti della libreria generale

101.1 os16: file isolati della directory «lib/»

101.1.1 lib/NULL.h

Si veda la sezione 97.2.

```
2030001 #ifndef _NULL_H
2030002 #define _NULL_H       1
2030003
2030004 #define NULL 0
2030005
2030006 #endif
```

101.1.2 lib/SEEK.h

«

Si veda la sezione 97.2.

```
2040001 #ifndef _SEEK_H
2040002 #define _SEEK_H       1
2040003
2040004 //----------------------------------------------------------------------
2040005 // These values are used inside <stdio.h> and <unistd.h>
2040006 //----------------------------------------------------------------------
2040007 #define SEEK_SET        0   // From the start.
2040008 #define SEEK_CUR        1   // From current position.
2040009 #define SEEK_END        2   // From the end.
2040010 //----------------------------------------------------------------------
2040011
2040012 #endif
```

101.1.3 lib/_Bool.h

«

Si veda la sezione 97.2.

```
2050001 #ifndef __BOOL_H
2050002 #define __BOOL_H       1
2050003
2050004 typedef unsigned char _Bool;
2050005
2050006 #endif
```

101.1.4 lib/clock_t.h

«

Si veda la sezione 97.2.

```
2060001 #ifndef _CLOCK_T_H
2060002 #define _CLOCK_T_H       1
2060003
2060004 typedef unsigned long int clock_t;      // 32 bit unsigned int.
2060005
2060006 #endif
```

101.1.5 lib/const.h

«

Si veda la sezione 97.2.

```
2070001 #ifndef _CONST_H
2070002 #define _CONST_H       1
2070003
2070004 #define const
```

```
2070005
2070006 #endif
```

101.1.6 lib/ctype.h

Si veda la sezione 97.2.

```
2080001 #ifndef _CTYPE_H
2080002 #define _CTYPE_H       1
2080003 //----------------------------------------------------------------------
2080004
2080005 #include <NULL.h>
2080006 //----------------------------------------------------------------------
2080007 #define isblank(C)   ((int) (C == ' ' || C == '\t'))
2080008 #define isspace(C)   ((int) (C == ' ' \
2080009                              || C == '\f' \
2080010                              || C == '\n' \
2080011                              || C == '\n' \
2080012                              || C == '\r' \
2080013                              || C == '\t' \
2080014                              || C == '\v'))
2080015 #define isdigit(C)   ((int) (C >= '0' && C <= '9'))
2080016 #define isxdigit(C)  ((int) ((C >= '0' && C <= '9') \
2080017                               || (C >= 'A' && C <= 'F') \
2080018                               || (C >= 'a' && C <= 'f')))
2080019 #define isupper(C)   ((int) (C >= 'A' && C <= 'Z'))
2080020 #define islower(C)   ((int) (C >= 'a' && C <= 'z'))
2080021 #define iscntrl(C)   ((int) ((C >= 0x00 && C <= 0x1F) || C == 0x7F))
2080022 #define isgraph(C)   ((int) (C >= 0x21 && C <= 0x7E))
2080023 #define isprint(C)   ((int) (C >= 0x20 && C <= 0x7E))
2080024 #define isalpha(C)   (isupper (C) || islower (C))
2080025 #define isalnum(C)   (isalpha (C) || isdigit (C))
2080026 #define ispunct(C)   (isgraph (C) && (!isspace (C)) && (!isalnum (C)))
2080027 #define tolower(C)   (isupper (C) ? ((C) + 0x20) : (C))
2080028 #define toupper(C)   (islower (C) ? ((C) - 0x20) : (C))
2080029 #define toascii(C)   (C & 0x7F)
2080030 #define _tolower(C)  (isupper (C) ? ((C) + 0x20) : (C))
2080031 #define _toupper(C)  (islower (C) ? ((C) - 0x20) : (C))
2080032 //----------------------------------------------------------------------
2080033
2080034 #endif
```

101.1.7 lib/inttypes.h

«

Si veda la sezione 97.2.

```
2090001 #ifndef _INTTYPES_H
2090002 #define _INTTYPES_H     1
2090003 //----------------------------------------------------------------------
2090004 
2090005 #include <const.h>
2090006 #include <restrict.h>
2090007 #include <stdint.h>
2090008 #include <wchar_t.h>
2090009 //----------------------------------------------------------------------
2090010 typedef struct {
2090011     intmax_t quot;
2090012     intmax_t rem;
2090013 } imaxdiv_t;
2090014 //
2090015 imaxdiv_t imaxdiv    (intmax_t numer, intmax_t denom);
2090016 //----------------------------------------------------------------------
2090017 // Output typesetting.
2090018 //----------------------------------------------------------------------
2090019 #define PRId8           "d"
2090020 #define PRId16          "d"
2090021 #define PRId32          "ld"
2090022 #define PRIdLEAST8      "d"
2090023 #define PRIdLEAST16     "d"
2090024 #define PRIdLEAST32     "ld"
2090025 #define PRIdFAST8       "d"
2090026 #define PRIdFAST16      "d"
2090027 #define PRIdFAST32      "ld"
2090028 #define PRIdMAX         "ld"
2090029 #define PRIdPTR         "d"
2090030 #define PRIi8           "i"
2090031 #define PRIi16          "i"
2090032 #define PRIi32          "li"
2090033 #define PRIiLEAST8      "i"
2090034 #define PRIiLEAST16     "i"
2090035 #define PRIiLEAST32     "li"
2090036 #define PRIiFAST8       "i"
2090037 #define PRIiFAST16      "i"
2090038 #define PRIiFAST32      "i"
2090039 #define PRIiMAX         "li"
2090040 #define PRIiPTR         "i"
2090041 #define PRIo8           "o"
2090042 #define PRIo16          "o"
2090043 #define PRIo32          "lo"
2090044 #define PRIoLEAST8      "o"
2090045 #define PRIoLEAST16     "o"
```

```
2090046 #define PRIoLEAST32     "lo"
2090047 #define PRIoFAST8       "o"
2090048 #define PRIoFAST16      "o"
2090049 #define PRIoFAST32      "lo"
2090050 #define PRIoMAX         "lo"
2090051 #define PRIoPTR         "o"
2090052 #define PRIu8           "u"
2090053 #define PRIu16          "u"
2090054 #define PRIu32          "lu"
2090055 #define PRIuLEAST8      "u"
2090056 #define PRIuLEAST16     "u"
2090057 #define PRIuLEAST32     "lu"
2090058 #define PRIuFAST8       "u"
2090059 #define PRIuFAST16      "u"
2090060 #define PRIuFAST32      "lu"
2090061 #define PRIuMAX         "lu"
2090062 #define PRIuPTR         "u"
2090063 #define PRIx8           "x"
2090064 #define PRIx16          "x"
2090065 #define PRIx32          "lx"
2090066 #define PRIxLEAST8      "x"
2090067 #define PRIxLEAST16     "x"
2090068 #define PRIxLEAST32     "lx"
2090069 #define PRIxFAST8       "x"
2090070 #define PRIxFAST16      "x"
2090071 #define PRIxFAST32      "lx"
2090072 #define PRIxMAX         "lx"
2090073 #define PRIxPTR         "x"
2090074 #define PRIX8           "X"
2090075 #define PRIX16          "X"
2090076 #define PRIX32          "lX"
2090077 #define PRIXLEAST8      "X"
2090078 #define PRIXLEAST16     "X"
2090079 #define PRIXLEAST32     "lX"
2090080 #define PRIXFAST8       "X"
2090081 #define PRIXFAST16      "X"
2090082 #define PRIXFAST32      "lX"
2090083 #define PRIXMAX         "lX"
2090084 #define PRIXPTR         "X"
2090085 //----------------------------------------------------------------------
2090086 // Input scan and evaluation.
2090087 //----------------------------------------------------------------------
2090088 #define SCNd8           "hhd"
2090089 #define SCNd16          "hd"
2090090 #define SCNd32          "d"
2090091 #define SCNdLEAST8      "hhd"
2090092 #define SCNdLEAST16     "hd"
2090093 #define SCNdLEAST32     "d"
```

```
2090094 #define SCNdFAST8        "hhd"
2090095 #define SCNdFAST16       "d"
2090096 #define SCNdFAST32       "d"
2090097 #define SCNdMAX          "ld"
2090098 #define SCNdPTR          "d"
2090099 #define SCNi8            "hhi"
2090100 #define SCNi16           "hi"
2090101 #define SCNi32           "i"
2090102 #define SCNiLEAST8       "hhi"
2090103 #define SCNiLEAST16      "hi"
2090104 #define SCNiLEAST32      "i"
2090105 #define SCNiFAST8        "hhi"
2090106 #define SCNiFAST16       "i"
2090107 #define SCNiFAST32       "i"
2090108 #define SCNiMAX          "li"
2090109 #define SCNiPTR          "i"
2090110 #define SCNo8            "hho"
2090111 #define SCNo16           "ho"
2090112 #define SCNo32           "o"
2090113 #define SCNoLEAST8       "hho"
2090114 #define SCNoLEAST16      "ho"
2090115 #define SCNoLEAST32      "o"
2090116 #define SCNoFAST8        "hho"
2090117 #define SCNoFAST16       "o"
2090118 #define SCNoFAST32       "o"
2090119 #define SCNoMAX          "lo"
2090120 #define SCNoPTR          "o"
2090121 #define SCNu8            "hhu"
2090122 #define SCNu16           "hu"
2090123 #define SCNu32           "u"
2090124 #define SCNuLEAST8       "hhu"
2090125 #define SCNuLEAST16      "hu"
2090126 #define SCNuLEAST32      "u"
2090127 #define SCNuFAST8        "hhu"
2090128 #define SCNuFAST16       "u"
2090129 #define SCNuFAST32       "u"
2090130 #define SCNuMAX          "lu"
2090131 #define SCNuPTR          "u"
2090132 #define SCNx8            "hhx"
2090133 #define SCNx16           "hx"
2090134 #define SCNx32           "x"
2090135 #define SCNxLEAST8       "hhx"
2090136 #define SCNxLEAST16      "hx"
2090137 #define SCNxLEAST32      "x"
2090138 #define SCNxFAST8        "hhx"
2090139 #define SCNxFAST16       "x"
2090140 #define SCNxFAST32       "x"
2090141 #define SCNxMAX          "lx"
```

```
2090142 #define SCNxPTR         "x"
2090143 //----------------------------------------------------------------------
2090144 intmax_t  strtoimax  (const char *restrict nptr,
2090145                        char **restrict endptr, int base);
2090146 uintmax_t strtouimax (const char *restrict nptr,
2090147                        char **restrict endptr, int base);
2090148 intmax_t  wcstoimax  (const wchar_t *restrict nptr,
2090149                        wchar_t **restrict endptr, int base);
2090150 uintmax_t wcstouimax (const wchar_t *restrict nptr,
2090151                        wchar_t **restrict endptr, int base);
2090152 //----------------------------------------------------------------------
2090153 
2090154 #endif
```

101.1.8 lib/limits.h

Si veda la sezione 97.2.

```
2100001 #ifndef _LIMITS_H
2100002 #define _LIMITS_H       1
2100003 //----------------------------------------------------------------------
2100004 #define CHAR_BIT                (8)
2100005 #define SCHAR_MIN            (-0x80)
2100006 #define SCHAR_MAX             (0x7F)
2100007 #define UCHAR_MAX             (0xFF)
2100008 #define CHAR_MIN        SCHAR_MIN
2100009 #define CHAR_MAX        SCHAR_MAX
2100010 #define MB_LEN_MAX              (1)
2100011 #define SHRT_MIN           (-0x8000)
2100012 #define SHRT_MAX            (0x7FFF)
2100013 #define USHRT_MAX           (0xFFFF)
2100014 #define INT_MIN            (-0x8000)
2100015 #define INT_MAX             (0x7FFF)
2100016 #define UINT_MAX            (0xFFFFU)
2100017 #define LONG_MIN      (-0x80000000L)
2100018 #define LONG_MAX       (0x7FFFFFFFL)
2100019 #define ULONG_MAX      (0xFFFFFFFFUL)
2100020 //----------------------------------------------------------------------
2100021 #define LLONG_MIN     (-0x80000000L)     // The type 'long long int'
2100022 #define LLONG_MAX      (0x7FFFFFFFL)     // is not available with
2100023 #define ULLONG_MAX     (0xFFFFFFFFUL)    // a K&R C compiler.
2100024 //----------------------------------------------------------------------
2100025 #define WORD_BIT                16  // POSIX requires at least 32!
2100026 #define LONG_BIT                32
2100027 #define SSIZE_MAX         LONG_MAX
2100028 //----------------------------------------------------------------------
2100029 #define ARG_MAX               1024  // Arguments+environment max length.
2100030 #define ATEXIT_MAX              32  // Max "at exit" functions.
```

```
2100031 #define FILESIZEBITS          32  // File size needs integer size...
2100032 #define LINK_MAX             254  // Max links per file.
2100033 #define NAME_MAX              14  // File name max (Minix 1 fs).
2100034 #define OPEN_MAX               8  // Max open files per process.
2100035 #define PATH_MAX              64  // Path, including final '\0'.
2100036 #define MAX_CANON              1  // Max bytes in canonical tty queue.
2100037 #define MAX_INPUT              1  // Max bytes in tty input queue.
2100038 //----------------------------------------------------------------------
2100039 #define CHLD_MAX         INT_MAX  // Not used.
2100040 #define HOST_NAME_MAX    INT_MAX  // Not used.
2100041 #define LOGIN_NAME_MAX   INT_MAX  // Not used.
2100042 #define PAGE_SIZE        INT_MAX  // Not used.
2100043 #define RE_DUP_MAX       INT_MAX  // Not used.
2100044 #define STREAM_MAX       INT_MAX  // Not used.
2100045 #define SYMLOOP_MAX      INT_MAX  // Not used.
2100046 #define TTY_NAME_MAX     INT_MAX  // Not used.
2100047 #define TZNAME_MAX       INT_MAX  // Not used.
2100048 #define PIPE_MAX         INT_MAX  // Not used.
2100049 #define SYMLINK_MAX      INT_MAX  // Not used.
2100050 //----------------------------------------------------------------------
2100051
2100052 #endif
```

101.1.9 lib/ptrdiff_t.h

«

Si veda la sezione 97.2.

```
2110001 #ifndef _PTRDIFF_T_H
2110002 #define _PTRDIFF_T_H    1
2110003
2110004 typedef int ptrdiff_t;
2110005
2110006 #endif
```

101.1.10 lib/restrict.h

«

Si veda la sezione 97.2.

```
2120001 #ifndef _RESTRICT_H
2120002 #define _RESTRICT_H      1
2120003
2120004 #define restrict
2120005
2120006 #endif
```

101.1.11 lib/size_t.h

Si veda la sezione 97.2.

```
2130001 #ifndef _SIZE_T_H
2130002 #define _SIZE_T_H       1
2130003 //
2130004 // The type 'size_t' *must* be equal to an 'int'.
2130005 //
2130006 typedef unsigned int size_t;
2130007
2130008 #endif
```

101.1.12 lib/stdarg.h

Si veda la sezione 97.2.

```
2140001 #ifndef _STDARG_H
2140002 #define _STDARG_H       1
2140003 //----------------------------------------------------------------------
2140004 typedef unsigned char *va_list;
2140005 //----------------------------------------------------------------------
2140006
2140007 #define va_start(ap, last) ((void) ((ap) = \
2140008                              ((va_list) &(last)) + (sizeof (last))))
2140009 #define va_end(ap)          ((void) ((ap) = 0))
2140010 #define va_copy(dest, src) ((void) ((dest) = (va_list) (src)))
2140011 #define va_arg(ap, type)    (((ap) = (ap) + (sizeof (type))), \
2140012                              *((type *) ((ap) - (sizeof (type)))))
2140013 //----------------------------------------------------------------------
2140014
2140015 #endif
```

101.1.13 lib/stdbool.h

Si veda la sezione 97.2.

```
2150001 #ifndef _STDBOOL_H
2150002 #define _STDBOOL_H      1
2150003 //----------------------------------------------------------------------
2150004 typedef unsigned char bool;     // [1]
2150005 #define true    ((bool) 1)
2150006 #define false   ((bool) 0)
2150007 #define __bool_true_false_are_defined   1
2150008 //
2150009 // [1] For some reason, it cannot be defined as a macro expanding to
2150010 //     '_Bool'. Anyway, it is the same kind of type.
2150011 //
```

```
2150012 //----------------------------------------------------------------------
2150013
2150014 #endif
```

101.1.14 lib/stddef.h

Si veda la sezione 97.2.

```
2160001 #ifndef _STDDEF_H
2160002 #define _STDDEF_H      1
2160003 //----------------------------------------------------------------------
2160004
2160005 #include <ptrdiff_t.h>
2160006 #include <size_t.h>
2160007 #include <wchar_t.h>
2160008 #include <NULL.h>
2160009 //----------------------------------------------------------------------
2160010 #define offsetof(type, member)   ((size_t) &((type *)0)->member)
2160011 //----------------------------------------------------------------------
2160012
2160013 #endif
```

101.1.15 lib/stdint.h

Si veda la sezione 97.2.

```
2170001 #ifndef _STDINT_H
2170002 #define _STDINT_H       1
2170003 //----------------------------------------------------------------------
2170004 typedef signed char                  int8_t;
2170005 typedef short int                   int16_t;
2170006 typedef long int                    int32_t;
2170007 typedef unsigned char               uint8_t;
2170008 typedef unsigned short int         uint16_t;
2170009 typedef unsigned long int          uint32_t;
2170010 //
2170011 #define INT8_MIN                     (-0x80)
2170012 #define INT8_MAX                      (0x7F)
2170013 #define UINT8_MAX                     (0xFF)
2170014 #define INT16_MIN                  (-0x8000)
2170015 #define INT16_MAX                   (0x7FFF)
2170016 #define UINT16_MAX                  (0xFFFF)
2170017 #define INT32_MIN              (-0x80000000)
2170018 #define INT32_MAX               (0x7FFFFFFF)
2170019 #define UINT32_MAX              (0xFFFFFFFFU)
2170020 //----------------------------------------------------------------------
2170021 typedef signed char            int_least8_t;
```

```
2170022 typedef short int             int_least16_t;
2170023 typedef long int              int_least32_t;
2170024 typedef unsigned char         uint_least8_t;
2170025 typedef unsigned short int   uint_least16_t;
2170026 typedef unsigned long int    uint_least32_t;
2170027 //
2170028 #define INT_LEAST8_MIN                 (-0x80)
2170029 #define INT_LEAST8_MAX                  (0x7F)
2170030 #define UINT_LEAST8_MAX                 (0xFF)
2170031 #define INT_LEAST16_MIN               (-0x8000)
2170032 #define INT_LEAST16_MAX                (0x7FFF)
2170033 #define UINT_LEAST16_MAX               (0xFFFF)
2170034 #define INT_LEAST32_MIN           (-0x80000000)
2170035 #define INT_LEAST32_MAX            (0x7FFFFFFF)
2170036 #define UINT_LEAST32_MAX          (0xFFFFFFFFU)
2170037 //----------------------------------------------------------------------
2170038 #define INT8_C(VAL)                  VAL
2170039 #define INT16_C(VAL)                 VAL
2170040 #define INT32_C(VAL)                 VAL
2170041 #define UINT8_C(VAL)                 VAL
2170042 #define UINT16_C(VAL)                VAL
2170043 #define UINT32_C(VAL)                VAL ## U
2170044 //----------------------------------------------------------------------
2170045 typedef signed char              int_fast8_t;
2170046 typedef int                     int_fast16_t;
2170047 typedef long int                int_fast32_t;
2170048 typedef unsigned char           uint_fast8_t;
2170049 typedef unsigned int           uint_fast16_t;
2170050 typedef unsigned long int      uint_fast32_t;
2170051 //
2170052 #define INT_FAST8_MIN                  (-0x80)
2170053 #define INT_FAST8_MAX                   (0x7F)
2170054 #define UINT_FAST8_MAX                  (0xFF)
2170055 #define INT_FAST16_MIN            (-0x80000000)
2170056 #define INT_FAST16_MAX             (0x7FFFFFFF)
2170057 #define UINT_FAST16_MAX            (0xFFFFFFFFU)
2170058 #define INT_FAST32_MIN            (-0x80000000)
2170059 #define INT_FAST32_MAX             (0x7FFFFFFF)
2170060 #define UINT_FAST32_MAX            (0xFFFFFFFFU)
2170061 //----------------------------------------------------------------------
2170062 typedef int                          intptr_t;
2170063 typedef unsigned int                uintptr_t;
2170064 //
2170065 #define INTPTR_MIN                (-0x80000000)
2170066 #define INTPTR_MAX                (0x7FFFFFFF)
2170067 #define UINTPTR_MAX               (0xFFFFFFFFU)
2170068 //----------------------------------------------------------------------
2170069 typedef long int                    intmax_t;
```

```
2170070 typedef unsigned long int          uintmax_t;
2170071 //
2170072 #define INTMAX_C(VAL)              VAL ## L
2170073 #define UINTMAX_C(VAL)             VAL ## UL
2170074 //
2170075 #define INTMAX_MIN               (-0x80000000L)
2170076 #define INTMAX_MAX                (0x7FFFFFFFL)
2170077 #define UINTMAX_MAX               (0xFFFFFFFFUL)
2170078 //----------------------------------------------------------------------
2170079 #define PTRDIFF_MIN              (-0x80000000)
2170080 #define PTRDIFF_MAX               (0x7FFFFFFF)
2170081 //
2170082 #define SIG_ATOMIC_MIN           (-0x80000000)
2170083 #define SIG_ATOMIC_MAX            (0x7FFFFFFF)
2170084 //
2170085 #define SIZE_MAX                      (0xFFFFU)
2170086 //
2170087 #define WCHAR_MIN                          (0)
2170088 #define WCHAR_MAX                       (0xFFU)
2170089 //
2170090 #define WINT_MIN                       (-0x80L)
2170091 #define WINT_MAX                        (0x7FL)
2170092 //----------------------------------------------------------------------
2170093 
2170094 #endif
```

101.1.16 lib/time_t.h

Si veda la sezione 97.2.

```
2180001 #ifndef _TIME_T_H
2180002 #define _TIME_T_H       1
2180003 
2180004 typedef long int time_t;
2180005 
2180006 #endif
```

101.1.17 lib/wchar_t.h

Si veda la sezione 97.2.

```
2190001 #ifndef _WCHAR_T_H
2190002 #define _WCHAR_T_H      1
2190003 
2190004 typedef unsigned char wchar_t;
2190005 
2190006 #endif
```

101.2 os16: «lib/dirent.h»

Si veda la sezione 97.2.

```
2200001 #ifndef _DIRENT_H
2200002 #define _DIRENT_H       1
2200003
2200004 #include <sys/types.h>  // ino_t
2200005 #include <limits.h>     // NAME_MAX
2200006 #include <const.h>
2200007
2200008 //----------------------------------------------------------------------
2200009 struct dirent {
2200010     ino_t  d_ino;                   // I-node number [1]
2200011     char   d_name[NAME_MAX+1];      // NAME_MAX + Null termination
2200012 };
2200013 //
2200014 // [1] The type 'ino_t' must be equal to 'uint16_t', because the
2200015 //     directory inside the Minix 1 file system has exactly such
2200016 //     size.
2200017 //
2200018 //----------------------------------------------------------------------
2200019 #define DOPEN_MAX   OPEN_MAX/2  // <limits.h> [1]
2200020 //
2200021 // [1] DOPEN_MAX is not standard, but it is used to define how many
2200022 //     directory slot to keep for open directories. As directory streams
2200023 //     are opened as file descriptors, the sum of all kind of file open
2200024 //     cannot be more than OPEM_MAX.
2200025 //----------------------------------------------------------------------
2200026 typedef struct {
2200027     int            fdn;       // File descriptor number.
2200028     struct dirent dir;       // Last directory item read.
2200029 } DIR;
2200030
2200031 extern DIR _directory_stream[]; // Defined inside 'lib/dirent/DIR.c'.
2200032 //----------------------------------------------------------------------
2200033 // Function prototypes.
2200034 //----------------------------------------------------------------------
2200035 int             closedir  (DIR *dp);
2200036 DIR            *opendir   (const char *name);
2200037 struct dirent *readdir   (DIR *dp);
2200038 void            rewinddir (DIR *dp);
2200039 //----------------------------------------------------------------------
2200040
2200041 #endif
```

101.2.1 lib/dirent/DIR.c

«

Si veda la sezione 97.2.

```
2210001 #include <dirent.h>
2210002 //
2210003 // There must be room for at least 'DOPEN_MAX' elements.
2210004 //
2210005 DIR _directory_stream[DOPEN_MAX];
2210006
2210007 void
2210008 _dirent_directory_stream_setup (void)
2210009 {
2210010     int d;
2210011     //
2210012     for (d = 0; d < DOPEN_MAX; d++)
2210013       {
2210014         _directory_stream[d].fdn    = -1;
2210015       }
2210016 }
```

101.2.2 lib/dirent/closedir.c

«

Si veda la sezione 94.10.

```
2220001 #include <dirent.h>
2220002 #include <fcntl.h>
2220003 #include <const.h>
2220004 #include <sys/types.h>
2220005 #include <sys/stat.h>
2220006 #include <unistd.h>
2220007 #include <errno.h>
2220008 #include <stddef.h>
2220009 //----------------------------------------------------------------------
2220010 int
2220011 closedir (DIR *dp)
2220012 {
2220013     //
2220014     // Check for a valid argument
2220015     //
2220016     if (dp == NULL)
2220017       {
2220018         //
2220019         // Not a valid pointer.
2220020         //
2220021         errset (EBADF);          // Invalid directory.
2220022         return (-1);
2220023       }
2220024     //
```

```
2220025     // Check if it is an open directory stream.
2220026     //
2220027     if (dp->fdn < 0)
2220028       {
2220029         //
2220030         // The stream is closed.
2220031         //
2220032         errset (EBADF);          // Invalid directory.
2220033         return (-1);
2220034       }
2220035     //
2220036     // Close the file descriptor. It there is an error,
2220037     // the 'errno' variable will be set by 'close()'.
2220038     //
2220039     return (close (dp->fdn));
2220040 }
```

101.2.3 lib/dirent/opendir.c

Si veda la sezione 94.76.

```
2230001 #include <dirent.h>
2230002 #include <fcntl.h>
2230003 #include <const.h>
2230004 #include <sys/types.h>
2230005 #include <sys/stat.h>
2230006 #include <unistd.h>
2230007 #include <errno.h>
2230008 #include <stddef.h>
2230009 //----------------------------------------------------------------------
2230010 DIR *
2230011 opendir (const char *path)
2230012 {
2230013     int    fdn;
2230014     int    d;
2230015     DIR   *dp;
2230016     struct stat file_status;
2230017     //
2230018     // Function 'opendir()' is used only for reading.
2230019     //
2230020     fdn = open (path, O_RDONLY);
2230021     //
2230022     // Check the file descriptor returned.
2230023     //
2230024     if (fdn < 0)
2230025       {
2230026         //
2230027         // The variable 'errno' is already set:
```

```
2230028             //   EINVAL
2230029             //   EMFILE
2230030             //   ENFILE
2230031             //
2230032             errset (errno);
2230033             return (NULL);
2230034           }
2230035         //
2230036         // Set the 'FD_CLOEXEC' flag for that file descriptor.
2230037         //
2230038         if (fcntl (fdn, F_SETFD, FD_CLOEXEC) != 0)
2230039           {
2230040             //
2230041             // The variable 'errno' is already set:
2230042             //   EBADF
2230043             //
2230044             errset (errno);
2230045             close (fdn);
2230046             return (NULL);
2230047           }
2230048         //
2230049         //
2230050         //
2230051         if (fstat (fdn, &file_status) != 0)
2230052           {
2230053             //
2230054             // Error should be already set.
2230055             //
2230056             errset (errno);
2230057             close (fdn);
2230058             return (NULL);
2230059           }
2230060         //
2230061         // Verify it is a directory
2230062         //
2230063         if (!S_ISDIR(file_status.st_mode))
2230064           {
2230065             //
2230066             // It is not a directory!
2230067             //
2230068             close (fdn);
2230069             errset (ENOTDIR);                // Is not a directory.
2230070             return (NULL);
2230071           }
2230072         //
2230073         // A valid file descriptor is available: must find a free
2230074         // '_directory_stream[]' slot.
2230075         //
```

```
2230076     for (d = 0; d < DOPEN_MAX; d++)
2230077       {
2230078         if (_directory_stream[d].fdn < 0)
2230079           {
2230080             //
2230081             // Found a free slot: set it up.
2230082             //
2230083             dp = &(_directory_stream[d]);
2230084             dp->fdn    = fdn;
2230085             //
2230086             // Return the directory pointer.
2230087             //
2230088             return (dp);
2230089           }
2230090       }
2230091     //
2230092     // If we are here, there was no free directory slot available.
2230093     //
2230094     close  (fdn);
2230095     errset (EMFILE);                    // Too many file open.
2230096     return (NULL);
2230097 }
```

101.2.4 lib/dirent/readdir.c

Si veda la sezione 94.86.

```
2240001 #include <dirent.h>
2240002 #include <fcntl.h>
2240003 #include <sys/types.h>
2240004 #include <sys/stat.h>
2240005 #include <unistd.h>
2240006 #include <errno.h>
2240007 #include <stddef.h>
2240008 //----------------------------------------------------------------------
2240009 struct dirent *
2240010 readdir (DIR *dp)
2240011 {
2240012     ssize_t size;
2240013     //
2240014     // Check for a valid argument.
2240015     //
2240016     if (dp == NULL)
2240017       {
2240018         //
2240019         // Not a valid pointer.
2240020         //
2240021         errset (EBADF);         // Invalid directory.
```

```
2240022            return (NULL);
2240023          }
2240024        //
2240025        // Check if it is an open directory stream.
2240026        //
2240027        if (dp->fdn < 0)
2240028          {
2240029            //
2240030            // The stream is closed.
2240031            //
2240032            errset (EBADF);          // Invalid directory.
2240033            return (NULL);
2240034          }
2240035        //
2240036        // Read the directory.
2240037        //
2240038        size = read (dp->fdn, &(dp->dir),
2240039                     (size_t) 16);
2240040        //
2240041        // Fix the null termination, if the name is very long.
2240042        //
2240043        dp->dir.d_name[NAME_MAX] = '\0';
2240044        //
2240045        // Check what was read.
2240046        //
2240047        if (size == 0)
2240048          {
2240049            //
2240050            // End of directory, but it is not an error.
2240051            //
2240052            return (NULL);
2240053          }
2240054        //
2240055        if (size < 0)
2240056          {
2240057            //
2240058            // This is an error. The variable `errno' is already set.
2240059            //
2240060            errset (errno);
2240061            return (NULL);
2240062          }
2240063        //
2240064        if (dp->dir.d_ino == 0)
2240065          {
2240066            //
2240067            // This is a null directory record.
2240068            // Should try to read the next one.
2240069            //
```

```
2240070            return (readdir (dp));
2240071          }
2240072        //
2240073        if (strlen (dp->dir.d_name) == 0)
2240074          {
2240075            //
2240076            // This is a bad directory record: try to read next.
2240077            //
2240078            return (readdir (dp));
2240079          }
2240080        //
2240081        // A valid directory record should be available now.
2240082        //
2240083        return (&(dp->dir));
2240084    }
```

101.2.5 lib/dirent/rewinddir.c

«

Si veda la sezione 94.89.

```
2250001    #include <dirent.h>
2250002    #include <fcntl.h>
2250003    #include <const.h>
2250004    #include <sys/types.h>
2250005    #include <sys/stat.h>
2250006    #include <unistd.h>
2250007    #include <errno.h>
2250008    #include <stddef.h>
2250009    #include <stdio.h>
2250010    //------------------------------------------------------------------------
2250011    void
2250012    rewinddir (DIR *dp)
2250013    {
2250014        FILE *fp;
2250015        //
2250016        // Check for a valid argument.
2250017        //
2250018        if (dp == NULL)
2250019          {
2250020            //
2250021            // Nothing to rewind, and no error to set.
2250022            //
2250023            return;
2250024          }
2250025        //
2250026        // Check if it is an open directory stream.
2250027        //
2250028        if (dp->fdn < 0)
```

```
2250029         {
2250030           //
2250031           // The stream is closed.
2250032           // Nothing to rewind, and no error to set.
2250033           //
2250034           return;
2250035         }
2250036       //
2250037       //
2250038       //
2250039       fp = &_stream[dp->fdn];
2250040       //
2250041       rewind (fp);
2250042   }
```

101.3 os16: «lib/errno.h»

Si veda la sezione 94.18.

```
2260001 #ifndef _ERRNO_H
2260002 #define _ERRNO_H        1
2260003
2260004 #include <limits.h>
2260005 #include <string.h>
2260006 //----------------------------------------------------------------------
2260007 // The variable 'errno' is standard, but 'errln' and 'errfn' are added
2260008 // to keep track of the error source. Variable 'errln' is used to save
2260009 // the source file line number; variable 'errfn' is used to save the
2260010 // source file name. To set these variable in a consistent way it is
2260011 // also added a macro-instruction: 'errset'.
2260012 //----------------------------------------------------------------------
2260013 extern int   errno;
2260014 extern int   errln;
2260015 extern char  errfn[PATH_MAX];
2260016 #define errset(e)        (errln = __LINE__, \
2260017                           strncpy (errfn, __FILE__, PATH_MAX), \
2260018                           errno = e)
2260019 //----------------------------------------------------------------------
2260020 // Standard POSIX 'errno' macro variables.
2260021 //----------------------------------------------------------------------
2260022 #define E2BIG             1 // Argument list too long.
2260023 #define EACCES            2 // Permission denied.
2260024 #define EADDRINUSE        3 // Address in use.
2260025 #define EADDRNOTAVAIL     4 // Address not available.
2260026 #define EAFNOSUPPORT      5 // Address family not supported.
2260027 #define EAGAIN            6 // Resource unavailable, try again.
2260028 #define EALREADY          7 // Connection already in progress.
2260029 #define EBADF             8 // Bad file descriptor.
```

```
2260030 #define EBADMSG          9 // Bad message.
2260031 #define EBUSY           10 // Device or resource busy.
2260032 #define ECANCELED       11 // Operation canceled.
2260033 #define ECHILD          12 // No child processes.
2260034 #define ECONNABORTED    13 // Connection aborted.
2260035 #define ECONNREFUSED    14 // Connection refused.
2260036 #define ECONNRESET      15 // Connection reset.
2260037 #define EDEADLK         16 // Resource deadlock would occur.
2260038 #define EDESTADDRREQ    17 // Destination address required.
2260039 #define EDOM            18 // Mathematics argument out of domain of
2260040                            // function.
2260041 #define EDQUOT          19 // Reserved.
2260042 #define EEXIST          20 // File exists.
2260043 #define EFAULT          21 // Bad address.
2260044 #define EFBIG           22 // File too large.
2260045 #define EHOSTUNREACH    23 // Host is unreachable.
2260046 #define EIDRM           24 // Identifier removed.
2260047 #define EILSEQ          25 // Illegal byte sequence.
2260048 #define EINPROGRESS     26 // Operation in progress.
2260049 #define EINTR           27 // Interrupted function.
2260050 #define EINVAL          28 // Invalid argument.
2260051 #define EIO             29 // I/O error.
2260052 #define EISCONN         30 // Socket is connected.
2260053 #define EISDIR          31 // Is a directory.
2260054 #define ELOOP           32 // Too many levels of symbolic links.
2260055 #define EMFILE          33 // Too many open files.
2260056 #define EMLINK          34 // Too many links.
2260057 #define EMSGSIZE        35 // Message too large.
2260058 #define EMULTIHOP       36 // Reserved.
2260059 #define ENAMETOOLONG    37 // Filename too long.
2260060 #define ENETDOWN        38 // Network is down.
2260061 #define ENETRESET       39 // Connection aborted by network.
2260062 #define ENETUNREACH     40 // Network unreachable.
2260063 #define ENFILE          41 // Too many files open in system.
2260064 #define ENOBUFS         42 // No buffer space available.
2260065 #define ENODATA         43 // No message is available on the stream head
2260066                            // read queue.
2260067 #define ENODEV          44 // No such device.
2260068 #define ENOENT          45 // No such file or directory.
2260069 #define ENOEXEC         46 // Executable file format error.
2260070 #define ENOLCK          47 // No locks available.
2260071 #define ENOLINK         48 // Reserved.
2260072 #define ENOMEM          49 // Not enough space.
2260073 #define ENOMSG          50 // No message of the desired type.
2260074 #define ENOPROTOOPT     51 // Protocol not available.
2260075 #define ENOSPC          52 // No space left on device.
2260076 #define ENOSR           53 // No stream resources.
2260077 #define ENOSTR          54 // Not a stream.
```

```
2260078 #define ENOSYS          55 // Function not supported.
2260079 #define ENOTCONN        56 // The socket is not connected.
2260080 #define ENOTDIR         57 // Not a directory.
2260081 #define ENOTEMPTY       58 // Directory not empty.
2260082 #define ENOTSOCK        59 // Not a socket.
2260083 #define ENOTSUP         60 // Not supported.
2260084 #define ENOTTY          61 // Inappropriate I/O control operation.
2260085 #define ENXIO           62 // No such device or address.
2260086 #define EOPNOTSUPP      63 // Operation not supported on socket.
2260087 #define EOVERFLOW       64 // Value too large to be stored in data type.
2260088 #define EPERM           65 // Operation not permitted.
2260089 #define EPIPE           66 // Broken pipe.
2260090 #define EPROTO          67 // Protocol error.
2260091 #define EPROTONOSUPPORT 68 // Protocol not supported.
2260092 #define EPROTOTYPE      69 // Protocol wrong type for socket.
2260093 #define ERANGE          70 // Result too large.
2260094 #define EROFS           71 // Read-only file system.
2260095 #define ESPIPE          72 // Invalid seek.
2260096 #define ESRCH           73 // No such process.
2260097 #define ESTALE          74 // Reserved.
2260098 #define ETIME           75 // Stream ioctl() timeout.
2260099 #define ETIMEDOUT       76 // Connection timed out.
2260100 #define ETXTBSY         77 // Text file busy.
2260101 #define EWOULDBLOCK     78 // Operation would block
2260102                            // (may be the same as EAGAIN).
2260103 #define EXDEV           79 // Cross-device link.
2260104 //----------------------------------------------------------------------
2260105 // Added os16 errors.
2260106 //----------------------------------------------------------------------
2260107 #define EUNKNOWN                    (-1) // Unknown error.
2260108 #define E_FILE_TYPE                  80  // File type not compatible.
2260109 #define E_ROOT_INODE_NOT_CACHED      81  // The root directory inode is
2260110                                          // not cached.
2260111 #define E_CANNOT_READ_SUPERBLOCK     83  // Cannot read super block.
2260112 #define E_MAP_INODE_TOO_BIG          84  // Map inode too big.
2260113 #define E_MAP_ZONE_TOO_BIG           85  // Map zone too big.
2260114 #define E_DATA_ZONE_TOO_BIG          86  // Data zone too big.
2260115 #define E_CANNOT_FIND_ROOT_DEVICE    87  // Cannot find root device.
2260116 #define E_CANNOT_FIND_ROOT_INODE     88  // Cannot find root inode.
2260117 #define E_FILE_TYPE_UNSUPPORTED      89  // File type unsupported.
2260118 #define E_ENV_TOO_BIG                90  // Environment too big.
2260119 #define E_LIMIT                      91  // Exceeded implementation
2260120                                          // limits.
2260121 #define E_NOT_MOUNTED                92  // Not mounted.
2260122 #define E_NOT_IMPLEMENTED            93  // Not implemented.
2260123 //----------------------------------------------------------------------
2260124 // Default descriptions for errors.
2260125 //----------------------------------------------------------------------
```

```
2260126 #define TEXT_E2BIG              "Argument list too long."
2260127 #define TEXT_EACCES             "Permission denied."
2260128 #define TEXT_EADDRINUSE         "Address in use."
2260129 #define TEXT_EADDRNOTAVAIL      "Address not available."
2260130 #define TEXT_EAFNOSUPPORT       "Address family not supported."
2260131 #define TEXT_EAGAIN             "Resource unavailable, " \
2260132                                 "try again."
2260133 #define TEXT_EALREADY           "Connection already in " \
2260134                                 "progress."
2260135 #define TEXT_EBADF              "Bad file descriptor."
2260136 #define TEXT_EBADMSG            "Bad message."
2260137 #define TEXT_EBUSY              "Device or resource busy."
2260138 #define TEXT_ECANCELED          "Operation canceled."
2260139 #define TEXT_ECHILD             "No child processes."
2260140 #define TEXT_ECONNABORTED       "Connection aborted."
2260141 #define TEXT_ECONNREFUSED       "Connection refused."
2260142 #define TEXT_ECONNRESET         "Connection reset."
2260143 #define TEXT_EDEADLK            "Resource deadlock would occur."
2260144 #define TEXT_EDESTADDRREQ       "Destination address required."
2260145 #define TEXT_EDOM               "Mathematics argument out of " \
2260146                                 "domain of function."
2260147 #define TEXT_EDQUOT             "Reserved error: EDQUOT"
2260148 #define TEXT_EEXIST             "File exists."
2260149 #define TEXT_EFAULT             "Bad address."
2260150 #define TEXT_EFBIG              "File too large."
2260151 #define TEXT_EHOSTUNREACH       "Host is unreachable."
2260152 #define TEXT_EIDRM              "Identifier removed."
2260153 #define TEXT_EILSEQ             "Illegal byte sequence."
2260154 #define TEXT_EINPROGRESS        "Operation in progress."
2260155 #define TEXT_EINTR              "Interrupted function."
2260156 #define TEXT_EINVAL             "Invalid argument."
2260157 #define TEXT_EIO                "I/O error."
2260158 #define TEXT_EISCONN            "Socket is connected."
2260159 #define TEXT_EISDIR             "Is a directory."
2260160 #define TEXT_ELOOP              "Too many levels of " \
2260161                                 "symbolic links."
2260162 #define TEXT_EMFILE             "Too many open files."
2260163 #define TEXT_EMLINK             "Too many links."
2260164 #define TEXT_EMSGSIZE           "Message too large."
2260165 #define TEXT_EMULTIHOP          "Reserved error: EMULTIHOP"
2260166 #define TEXT_ENAMETOOLONG       "Filename too long."
2260167 #define TEXT_ENETDOWN           "Network is down."
2260168 #define TEXT_ENETRESET          "Connection aborted by network."
2260169 #define TEXT_ENETUNREACH        "Network unreachable."
2260170 #define TEXT_ENFILE             "Too many files open in system."
2260171 #define TEXT_ENOBUFS            "No buffer space available."
2260172 #define TEXT_ENODATA            "No message is available on " \
2260173                                 "the stream head read queue."
```

```
2260174 #define TEXT_ENODEV                         "No such device."
2260175 #define TEXT_ENOENT                         "No such file or directory."
2260176 #define TEXT_ENOEXEC                        "Executable file format error."
2260177 #define TEXT_ENOLCK                         "No locks available."
2260178 #define TEXT_ENOLINK                        "Reserved error: ENOLINK"
2260179 #define TEXT_ENOMEM                         "Not enough space."
2260180 #define TEXT_ENOMSG                         "No message of the desired " \
2260181                                             "type."
2260182 #define TEXT_ENOPROTOOPT                    "Protocol not available."
2260183 #define TEXT_ENOSPC                         "No space left on device."
2260184 #define TEXT_ENOSR                          "No stream resources."
2260185 #define TEXT_ENOSTR                         "Not a stream."
2260186 #define TEXT_ENOSYS                         "Function not supported."
2260187 #define TEXT_ENOTCONN                       "The socket is not connected."
2260188 #define TEXT_ENOTDIR                        "Not a directory."
2260189 #define TEXT_ENOTEMPTY                      "Directory not empty."
2260190 #define TEXT_ENOTSOCK                       "Not a socket."
2260191 #define TEXT_ENOTSUP                        "Not supported."
2260192 #define TEXT_ENOTTY                         "Inappropriate I/O control " \
2260193                                             "operation."
2260194 #define TEXT_ENXIO                          "No such device or address."
2260195 #define TEXT_EOPNOTSUPP                     "Operation not supported on " \
2260196                                             "socket."
2260197 #define TEXT_EOVERFLOW                      "Value too large to be " \
2260198                                             "stored in data type."
2260199 #define TEXT_EPERM                          "Operation not permitted."
2260200 #define TEXT_EPIPE                          "Broken pipe."
2260201 #define TEXT_EPROTO                         "Protocol error."
2260202 #define TEXT_EPROTONOSUPPORT                "Protocol not supported."
2260203 #define TEXT_EPROTOTYPE                     "Protocol wrong type for " \
2260204                                             "socket."
2260205 #define TEXT_ERANGE                         "Result too large."
2260206 #define TEXT_EROFS                          "Read-only file system."
2260207 #define TEXT_ESPIPE                         "Invalid seek."
2260208 #define TEXT_ESRCH                          "No such process."
2260209 #define TEXT_ESTALE                         "Reserved error: ESTALE"
2260210 #define TEXT_ETIME                          "Stream ioctl() timeout."
2260211 #define TEXT_ETIMEDOUT                      "Connection timed out."
2260212 #define TEXT_ETXTBSY                        "Text file busy."
2260213 #define TEXT_EWOULDBLOCK                    "Operation would block."
2260214 #define TEXT_EXDEV                          "Cross-device link."
2260215 //----------------------------------------------------------------------
2260216 #define TEXT_EUNKNOWN                       "Unknown error."
2260217 #define TEXT_E_FILE_TYPE                    "File type not compatible."
2260218 #define TEXT_E_ROOT_INODE_NOT_CACHED        "The root directory inode " \
2260219                                             "is not cached."
2260220 #define TEXT_E_CANNOT_READ_SUPERBLOCK       "Cannot read super block."
2260221 #define TEXT_E_MAP_INODE_TOO_BIG            "Map inode too big."
```

```
2260222 #define TEXT_E_MAP_ZONE_TOO_BIG         "Map zone too big."
2260223 #define TEXT_E_DATA_ZONE_TOO_BIG        "Data zone too big."
2260224 #define TEXT_E_CANNOT_FIND_ROOT_DEVICE  "Cannot find root device."
2260225 #define TEXT_E_CANNOT_FIND_ROOT_INODE   "Cannot find root inode."
2260226 #define TEXT_E_FILE_TYPE_UNSUPPORTED    "File type unsupported."
2260227 #define TEXT_E_ENV_TOO_BIG              "Environment too big."
2260228 #define TEXT_E_LIMIT                    "Exceeded implementation " \
2260229                                         "limits."
2260230 #define TEXT_E_NOT_MOUNTED               "Not mounted."
2260231 #define TEXT_E_NOT_IMPLEMENTED          "Not implemented."
2260232
2260233 //----------------------------------------------------------------------
2260234 // The function 'error()' is not standard and is used to return a
2260235 // pointer to a string containing the default description of the
2260236 // error contained inside 'errno'.
2260237 //----------------------------------------------------------------------
2260238 char *error      (void);        // Not standard!
2260239
2260240 #endif
```

101.3.1 lib/errno/errno.c

Si veda la sezione 94.18.

```
2270001 //----------------------------------------------------------------------
2270002 // This file does not include the 'errno.h' header, because here 'errno'
2270003 // should not be declared as an extern variable!
2270004 //----------------------------------------------------------------------
2270005
2270006 #include <limits.h>
2270007 //----------------------------------------------------------------------
2270008 // The variable 'errno' is standard, but 'errln' and 'errfn' are added
2270009 // to keep track of the error source. Variable 'errln' is used to save
2270010 // the source file line number; variable 'errfn' is used to save the
2270011 // source file name. To set these variable in a consistent way it is
2270012 // also added a macro-instruction: 'errset'.
2270013 //----------------------------------------------------------------------
2270014 int  errno;
2270015 int  errln;
2270016 char errfn[PATH_MAX];
2270017 //----------------------------------------------------------------------
```

101.4 os16: «lib/fcntl.h»

Si veda la sezione 97.2.

```
2280001 #ifndef _FCNTL_H
2280002 #define _FCNTL_H        1
2280003 
2280004 #include <const.h>
2280005 #include <sys/types.h>  // mode_t
2280006                         // off_t
2280007                         // pid_t
2280008 //----------------------------------------------------------------------
2280009 // Values for the second parameter of function 'fcntl()'.
2280010 //----------------------------------------------------------------------
2280011 #define F_DUPFD         0       // Duplicate file descriptor.
2280012 #define F_GETFD         1       // Get file descriptor flags.
2280013 #define F_SETFD         2       // Set file descriptor flags.
2280014 #define F_GETFL         3       // Get file status flags.
2280015 #define F_SETFL         4       // Set file status flags.
2280016 #define F_GETLK         5       // Get record locking information.
2280017 #define F_SETLK         6       // Set record locking information.
2280018 #define F_SETLKW        7       // Set record locking information;
2280019                                 // wait if blocked.
2280020 #define F_GETOWN        8       // Set owner of socket.
2280021 #define F_SETOWN        9       // Get owner of socket.
2280022 //----------------------------------------------------------------------
2280023 // Flags to be set with:
2280024 //   fcntl (fd, F_SETFD, ...);
2280025 //----------------------------------------------------------------------
2280026 #define FD_CLOEXEC      1       // Close the file descriptor upon
2280027                                 // execution of an exec() family
2280028                                 // function.
2280029 //----------------------------------------------------------------------
2280030 // Values for type 'l_type', used for record locking with 'fcntl()'.
2280031 //----------------------------------------------------------------------
2280032 #define F_RDLCK         0       // Read lock.
2280033 #define F_WRLCK         1       // Write lock.
2280034 #define F_UNLCK         2       // Remove lock.
2280035 //----------------------------------------------------------------------
2280036 // Flags for file creation, in place of 'oflag' parameter for function
2280037 // 'open()'.
2280038 //----------------------------------------------------------------------
2280039 #define O_CREAT         000010  // Create file if it does not exist.
2280040 #define O_EXCL          000020  // Exclusive use flag.
2280041 #define O_NOCTTY        000040  // Do not assign a controlling terminal.
2280042 #define O_TRUNC         000100  // Truncation flag.
2280043 //----------------------------------------------------------------------
2280044 // Flags for the file status, used with 'open()' and 'fcntl()'.
2280045 //----------------------------------------------------------------------
```

```
2280046 #define O_APPEND        000200  // Write append.
2280047 #define O_DSYNC         000400  // Synchronized write operations.
2280048 #define O_NONBLOCK      001000  // Non-blocking mode.
2280049 #define O_RSYNC         002000  // Synchronized read operations.
2280050 #define O_SYNC          004000  // Synchronized read and write.
2280051 //----------------------------------------------------------------------
2280052 // File access mask selection.
2280053 //----------------------------------------------------------------------
2280054 #define O_ACCMODE       000003  // Mask to select the last three bits,
2280055                                 // used to specify the main access
2280056                                 // modes: read, write and both.
2280057 //----------------------------------------------------------------------
2280058 // Main access modes.
2280059 //----------------------------------------------------------------------
2280060 #define O_RDONLY        000001                  // Read.
2280061 #define O_WRONLY        000002                  // Write.
2280062 #define O_RDWR          (O_RDONLY | O_WRONLY)   // Both read and write.
2280063 //----------------------------------------------------------------------
2280064 // Structure 'flock', used to file lock for POSIX standard. It is not
2280065 // used inside os16.
2280066 //----------------------------------------------------------------------
2280067 struct flock {
2280068     short int l_type;   // Type of lock: F_RDLCK, F_WRLCK, or F_UNLCK.
2280069     short int l_whence; // Start reference point.
2280070     off_t     l_start;  // Offset, from 'l_whence', for the area start.
2280071     off_t     l_len;    // Locked area size. Zero means up to the end of
2280072                         // the file.
2280073     pid_t     l_pid;    // The process id blocking the area.
2280074 };
2280075 //----------------------------------------------------------------------
2280076 // Function prototypes.
2280077 //----------------------------------------------------------------------
2280078 int creat (const char *path, mode_t mode);
2280079 int fcntl (int fdn, int cmd, ...);
2280080 int open  (const char *path, int oflags, ...);
2280081 //----------------------------------------------------------------------
2280082 
2280083 #endif
```

101.4.1 lib/fcntl/creat.c

Si veda la sezione 94.11.

```
2290001 #include <fcntl.h>
2290002 #include <sys/types.h>
2290003 #include <const.h>
2290004 //----------------------------------------------------------------------
2290005 int
```

```
2290006 creat (const char *path, mode_t mode)
2290007 {
2290008     return (open (path, O_WRONLY|O_CREAT|O_TRUNC, mode));
2290009 }
```

101.4.2 lib/fcntl/fcntl.c

«

Si veda la sezione 93.13.

```
2300001 #include <fcntl.h>
2300002 #include <stdarg.h>
2300003 #include <stddef.h>
2300004 #include <string.h>
2300005 #include <errno.h>
2300006 #include <sys/os16.h>
2300007 #include <const.h>
2300008 #include <limits.h>
2300009 //----------------------------------------------------------------------
2300010 int
2300011 fcntl (int fdn, int cmd, ...)
2300012 {
2300013     va_list ap;
2300014     sysmsg_fcntl_t msg;
2300015     va_start (ap, cmd);
2300016     //
2300017     // Well known arguments.
2300018     //
2300019     msg.fdn = fdn;
2300020     msg.cmd = cmd;
2300021     //
2300022     // Select other arguments.
2300023     //
2300024     switch (cmd)
2300025       {
2300026         case F_DUPFD:
2300027         case F_SETFD:
2300028         case F_SETFL:
2300029             msg.arg = va_arg (ap, int);
2300030             break;
2300031         case F_GETFD:
2300032         case F_GETFL:
2300033             break;
2300034         case F_GETOWN:
2300035         case F_SETOWN:
2300036         case F_GETLK:
2300037         case F_SETLK:
2300038         case F_SETLKW:
2300039             errset (E_NOT_IMPLEMENTED);         // Not implemented.
```

```
2300040                 return (-1);
2300041             default:
2300042                 errset (EINVAL);                     // Not implemented.
2300043                 return (NULL);
2300044           }
2300045         //
2300046         // Do the system call.
2300047         //
2300048         sys (SYS_FCNTL, &msg, (sizeof msg));
2300049         errno = msg.errno;
2300050         errln = msg.errln;
2300051         strncpy (errfn, msg.errfn, PATH_MAX);
2300052         return (msg.ret);
2300053     }
```

101.4.3 lib/fcntl/open.c

«

Si veda la sezione 93.28.

```
2310001 #include <fcntl.h>
2310002 #include <stdarg.h>
2310003 #include <stddef.h>
2310004 #include <string.h>
2310005 #include <errno.h>
2310006 #include <sys/os16.h>
2310007 #include <const.h>
2310008 #include <limits.h>
2310009 //----------------------------------------------------------------------
2310010 int
2310011 open (const char *path, int oflags, ...)
2310012 {
2310013     va_list ap;
2310014     sysmsg_open_t msg;
2310015     va_start (ap, oflags);
2310016     if (path == NULL || strlen (path) == 0)
2310017       {
2310018         errset (EINVAL);          // Invalid argument.
2310019         return (-1);
2310020       }
2310021     strncpy (msg.path, path, PATH_MAX);
2310022     msg.flags = oflags;
2310023     msg.mode  = va_arg (ap, mode_t);
2310024     sys (SYS_OPEN, &msg, (sizeof msg));
2310025     errno = msg.errno;
2310026     errln = msg.errln;
2310027     strncpy (errfn, msg.errfn, PATH_MAX);
2310028     return (msg.ret);
2310029 }
```

101.5 os16: «lib/grp.h»

Si veda la sezione 97.2.

```
2320001 //----------------------------------------------------------------------
2320002 // os16 does not have a group management!
2320003 //----------------------------------------------------------------------
2320004
2320005 #ifndef _GRP_H
2320006 #define _GRP_H        1
2320007
2320008 #include <const.h>
2320009 #include <restrict.h>
2320010 #include <sys/types.h>          // gid_t
2320011
2320012 //----------------------------------------------------------------------
2320013 struct group {
2320014     char   *gr_name;
2320015     gid_t   gr_gid;
2320016     char  **gr_mem;
2320017 };
2320018 //----------------------------------------------------------------------
2320019 struct group *getgrgid (gid_t gid);
2320020 struct group *getgrnam (const char *name);
2320021 //----------------------------------------------------------------------
2320022
2320023 #endif
```

101.5.1 lib/grp/getgrgid.c

Si veda la sezione 97.2.

```
2330001 #include <grp.h>
2330002 #include <NULL.h>
2330003 //----------------------------------------------------------------------
2330004 struct group *
2330005 getgrgid (gid_t gid)
2330006 {
2330007     static char *name = "none";
2330008     static struct group grp;
2330009     //
2330010     // os16 does not have a group management, so the answare is always
2330011     // the same.
2330012     //
2330013     grp.gr_name = name;
2330014     grp.gr_gid  = (gid_t) -1;
2330015     grp.gr_mem  = NULL;
2330016     //
2330017     return (&grp);
```

```
2330018 }
```

101.5.2 lib/grp/getgrnam.c

Si veda la sezione 97.2.

```
2340001 #include <grp.h>
2340002 //----------------------------------------------------------------------
2340003 struct group *
2340004 getgrnam (const char *name)
2340005 {
2340006     return (getgrgid ((gid_t) 0));
2340007 }
```

101.6 os16: «lib/libgen.h»

Si veda la sezione 97.2.

```
2350001 #ifndef _LIBGEN_H
2350002 #define _LIBGEN_H       1
2350003 
2350004 //----------------------------------------------------------------------
2350005 char *basename (char *path);
2350006 char *dirname  (char *path);
2350007 //----------------------------------------------------------------------
2350008 
2350009 #endif
```

101.6.1 lib/libgen/basename.c

Si veda la sezione 94.7.

```
2360001 #include <libgen.h>
2360002 #include <limits.h>
2360003 #include <stddef.h>
2360004 #include <string.h>
2360005 //----------------------------------------------------------------------
2360006 char *
2360007 basename (char *path)
2360008 {
2360009     static char  *point = ".";          // When 'path' is NULL.
2360010            char  *p;                    // Pointer inside 'path'.
2360011            int    i;                    // Scan index inside 'path'.
2360012     //
2360013     // Empty path.
2360014     //
```

```
2360015     if (path == NULL || strlen (path) == 0)
2360016       {
2360017         return (point);
2360018       }
2360019     //
2360020     // Remove all final `/' if it exists, excluded the first character:
2360021     // `i' is kept greater than zero.
2360022     //
2360023     for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
2360024       {
2360025         path[i] = 0;
2360026       }
2360027     //
2360028     // After removal of extra final `/', if there is only one `/', this
2360029     // is to be returned.
2360030     //
2360031     if (strncmp (path, "/",  PATH_MAX) == 0)
2360032       {
2360033         return (path);
2360034       }
2360035     //
2360036     // If there are no `/'.
2360037     //
2360038     if (strchr (path, '/') == NULL)
2360039       {
2360040         return (path);
2360041       }
2360042     //
2360043     // Find the last `/' and calculate a pointer to the base name.
2360044     //
2360045     p = strrchr (path, (unsigned int) '/');
2360046     p++;
2360047     //
2360048     // Return the pointer to the base name.
2360049     //
2360050     return (p);
2360051 }
```

101.6.2 lib/libgen/dirname.c

Si veda la sezione 94.7.

```
2370001 #include <libgen.h>
2370002 #include <limits.h>
2370003 #include <stddef.h>
2370004 #include <string.h>
2370005 //----------------------------------------------------------------------
2370006 char *
```

```
2370007 dirname (char *path)
2370008 {
2370009     static char  *point = ".";           // When 'path' is NULL.
2370010            char  *p;                     // Pointer inside 'path'.
2370011            int    i;                     // Scan index inside 'path'.
2370012     //
2370013     // Empty path.
2370014     //
2370015     if      (path == NULL || strlen (path) == 0)
2370016       {
2370017         return (point);
2370018       }
2370019     //
2370020     // Simple cases.
2370021     //
2370022     if (strncmp (path, "/",  PATH_MAX) == 0  ||
2370023         strncmp (path, ".",  PATH_MAX) == 0  ||
2370024         strncmp (path, "..", PATH_MAX) == 0)
2370025       {
2370026         return (path);
2370027       }
2370028     //
2370029     // Remove all final '/' if it exists, excluded the first character:
2370030     // 'i' is kept greater than zero.
2370031     //
2370032     for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
2370033       {
2370034         path[i] = 0;
2370035       }
2370036     //
2370037     // After removal of extra final '/', if there is only one '/', this
2370038     // is to be returned.
2370039     //
2370040     if (strncmp (path, "/",  PATH_MAX) == 0)
2370041       {
2370042         return (path);
2370043       }
2370044     //
2370045     // If there are no '/'
2370046     //
2370047     if (strchr (path, '/') == NULL)
2370048       {
2370049         return (point);
2370050       }
2370051     //
2370052     // If there is only a '/' a the beginning.
2370053     //
2370054     if (path[0] == '/'                                  &&
```

```
2370055             strchr (&path[1], (unsigned int) '/') == NULL)
2370056           {
2370057             path[1] = 0;
2370058             return (path);
2370059           }
2370060         //
2370061         // Replace the last '/' with zero.
2370062         //
2370063          p = strrchr (path, (unsigned int) '/');
2370064         *p = 0;
2370065         //
2370066         // Now remove extra duplicated final '/', except the very first
2370067         // character: 'i' is kept greater than zero.
2370068         //
2370069         for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
2370070           {
2370071             path[i] = 0;
2370072           }
2370073         //
2370074         // Now 'path' appears as a reduced string: the original path string
2370075         // is modified.
2370076         //
2370077         return (path);
2370078     }
```

101.7 os16: «lib/pwd.h»

Si veda la sezione 97.2.

```
2380001 #ifndef _PWD_H
2380002 #define _PWD_H        1
2380003 
2380004 #include <const.h>
2380005 #include <restrict.h>
2380006 #include <sys/types.h>          // gid_t, uid_t
2380007 //----------------------------------------------------------------------
2380008 struct passwd {
2380009     char   *pw_name;
2380010     char   *pw_passwd;
2380011     uid_t   pw_uid;
2380012     gid_t   pw_gid;
2380013     char   *pw_gecos;
2380014     char   *pw_dir;
2380015     char   *pw_shell;
2380016 };
2380017 //----------------------------------------------------------------------
2380018 struct passwd *getpwent (void);
2380019 void           setpwent (void);
```

```
2380020 void           endpwent (void);
2380021 struct passwd *getpwnam (const char *name);
2380022 struct passwd *getpwuid (uid_t uid);
2380023 //----------------------------------------------------------------------
2380024
2380025 #endif
```

101.7.1 lib/pwd/pwent.c

Si veda la sezione 94.53.

```
2390001 #include <pwd.h>
2390002 #include <stdio.h>
2390003 #include <string.h>
2390004 #include <stdlib.h>
2390005
2390006 //----------------------------------------------------------------------
2390007 static char          buffer[BUFSIZ];
2390008 static struct passwd pw;
2390009 static FILE         *fp = NULL;
2390010 //----------------------------------------------------------------------
2390011 struct passwd *
2390012 getpwent (void)
2390013 {
2390014     void *pstatus;
2390015     char *char_uid;
2390016     char *char_gid;
2390017     //
2390018     if (fp == NULL)
2390019       {
2390020         fp = fopen ("/etc/passwd", "r");
2390021         if (fp == NULL)
2390022           {
2390023             return NULL;
2390024           }
2390025       }
2390026     //
2390027     pstatus = fgets (buffer, BUFSIZ, fp);
2390028     if (pstatus == NULL)
2390029       {
2390030         return (NULL);
2390031       }
2390032     //
2390033     pw.pw_name   = strtok (buffer, ":");
2390034     pw.pw_passwd = strtok (NULL, ":");
2390035     char_uid     = strtok (NULL, ":");
2390036     char_gid     = strtok (NULL, ":");
2390037     pw.pw_gecos  = strtok (NULL, ":");
```

```
2390038     pw.pw_dir    = strtok (NULL, ":");
2390039     pw.pw_shell  = strtok (NULL, ":");
2390040     pw.pw_uid    = (uid_t) atoi (char_uid);
2390041     pw.pw_gid    = (gid_t) atoi (char_gid);
2390042     //
2390043     return (&pw);
2390044 }
2390045 //----------------------------------------------------------------------
2390046 void
2390047 endpwent (void)
2390048 {
2390049     int status;
2390050     //
2390051     if (fp != NULL)
2390052       {
2390053         fclose (fp);
2390054         if (status != NULL)
2390055           {
2390056             fp = NULL;
2390057           }
2390058       }
2390059 }
2390060 //----------------------------------------------------------------------
2390061 void
2390062 setpwent (void)
2390063 {
2390064     if (fp != NULL)
2390065       {
2390066         rewind (fp);
2390067       }
2390068 }
2390069 //----------------------------------------------------------------------
2390070 struct passwd *
2390071 getpwnam (const char *name)
2390072 {
2390073     struct passwd *pw;
2390074     //
2390075     setpwent ();
2390076     //
2390077     for (;;)
2390078       {
2390079         pw = getpwent ();
2390080         if (pw == NULL)
2390081           {
2390082             return (NULL);
2390083           }
2390084         if (strcmp (pw->pw_name, name) == 0)
2390085           {
```

```
2390086                 return (pw);
2390087               }
2390088           }
2390089 }
2390090 //----------------------------------------------------------------------
2390091 struct passwd *
2390092 getpwuid (uid_t uid)
2390093 {
2390094     struct passwd *pw;
2390095     //
2390096     setpwent ();
2390097     //
2390098     for (;;)
2390099       {
2390100         pw = getpwent ();
2390101         if (pw == NULL)
2390102           {
2390103             return (NULL);
2390104           }
2390105         if (pw->pw_uid == uid)
2390106           {
2390107             return (pw);
2390108           }
2390109       }
2390110 }
```

101.8 os16: «lib/signal.h»

«

Si veda la sezione 97.2.

```
2400001 #ifndef _SIGNAL_H
2400002 #define _SIGNAL_H       1
2400003 
2400004 #include <sys/types.h>
2400005 //----------------------------------------------------------------------
2400006 #define SIGHUP           1
2400007 #define SIGINT           2
2400008 #define SIGQUIT          3
2400009 #define SIGILL           4
2400010 #define SIGABRT          6
2400011 #define SIGFPE           8
2400012 #define SIGKILL          9
2400013 #define SIGSEGV         11
2400014 #define SIGPIPE         13
2400015 #define SIGALRM         14
2400016 #define SIGTERM         15
2400017 #define SIGSTOP         17
2400018 #define SIGTSTP         18
```

```
2400019 #define SIGCONT        19
2400020 #define SIGCHLD        20
2400021 #define SIGTTIN        21
2400022 #define SIGTTOU        22
2400023 #define SIGUSR1        30
2400024 #define SIGUSR2        31
2400025 //----------------------------------------------------------------------
2400026 typedef int sig_atomic_t;
2400027 typedef void (*sighandler_t) (int); // The type 'sighandler_t' is a
2400028                                     // pointer to a function for the
2400029                                     // signal handling, with a parameter
2400030                                     // of type 'int', returning 'void'.
2400031 //
2400032 // Special undeclarable functions.
2400033 //
2400034 #define SIG_ERR ((sighandler_t) -1) // It transform an integer number
2400035 #define SIG_DFL ((sighandler_t) 0)  // into a 'sighandler_t' type,
2400036 #define SIG_IGN ((sighandler_t) 1)  // that is, a pointer to a function
2400037                                     // that does not exists really.
2400038 //----------------------------------------------------------------------
2400039 sighandler_t signal (int sig, sighandler_t handler);
2400040 int          kill   (pid_t pid, int sig);
2400041 int          raise  (int sig);
2400042 //----------------------------------------------------------------------
2400043 
2400044 #endif
```

101.8.1 lib/signal/kill.c

Si veda la sezione 93.22.

```
2410001 #include <sys/os16.h>
2410002 #include <sys/types.h>
2410003 #include <signal.h>
2410004 #include <errno.h>
2410005 #include <string.h>
2410006 //----------------------------------------------------------------------
2410007 int
2410008 kill (pid_t pid, int sig)
2410009 {
2410010     sysmsg_kill_t msg;
2410011     if (pid < -1)       // Currently unsupported.
2410012       {
2410013         errset (ESRCH);
2410014         return (-1);
2410015       }
2410016     msg.pid    = pid;
2410017     msg.signal = sig;
```

```
2410018     msg.ret    = 0;
2410019     msg.errno  = 0;
2410020     sys (SYS_KILL, &msg, (sizeof msg));
2410021     errno = msg.errno;
2410022     errln = msg.errln;
2410023     strncpy (errfn, msg.errfn, PATH_MAX);
2410024     return (msg.ret);
2410025 }
```

101.8.2 lib/signal/signal.c

Si veda la sezione 93.34.

```
2420001 #include <sys/os16.h>
2420002 #include <sys/types.h>
2420003 #include <signal.h>
2420004 #include <errno.h>
2420005 #include <string.h>
2420006 //----------------------------------------------------------------------
2420007 sighandler_t
2420008 signal (int sig, sighandler_t handler)
2420009 {
2420010     sysmsg_signal_t msg;
2420011 
2420012     msg.signal  = sig;
2420013     msg.handler = handler;
2420014     msg.ret     = SIG_DFL;
2420015     msg.errno   = 0;
2420016     sys (SYS_SIGNAL, &msg, (sizeof msg));
2420017     errno = msg.errno;
2420018     errln = msg.errln;
2420019     strncpy (errfn, msg.errfn, PATH_MAX);
2420020     return (msg.ret);
2420021 }
```

101.9 os16: «lib/stdio.h»

Si veda la sezione 94.103.

```
2430001 #ifndef _STDIO_H
2430002 #define _STDIO_H        1
2430003 
2430004 #include <const.h>
2430005 #include <restrict.h>
2430006 #include <stdarg.h>
2430007 #include <stdint.h>
2430008 #include <limits.h>
```

```
2430009 #include <NULL.h>
2430010 #include <size_t.h>
2430011 #include <sys/types.h>
2430012 #include <SEEK.h>       // SEEK_CUR, SEEK_SET, SEEK_END
2430013 //----------------------------------------------------------------------
2430014 #define BUFSIZ              2048  // Like the file system max zone
2430015                                   // size.
2430016 #define _IOFBF                 0  // Input-output fully buffered.
2430017 #define _IOLBF                 1  // Input-output line buffered.
2430018 #define _IONBF                 2  // Input-output with no buffering.
2430019 
2430020 #define L_tmpnam    FILENAME_MAX  // <limits.h>
2430021 
2430022 #define FOPEN_MAX       OPEN_MAX  // <limits.h>
2430023 #define FILENAME_MAX    NAME_MAX  // <limits.h>
2430024 #define TMP_MAX           0x7FFF
2430025 
2430026 #define EOF                 (-1)  // Must be a negative value.
2430027 //----------------------------------------------------------------------
2430028 typedef off_t     fpos_t;    // 'off_t' defined in <sys/types.h>.
2430029 
2430030 typedef struct {
2430031     int           fdn;       // File descriptor number.
2430032     char          error;     // Error indicator.
2430033     char          eof;       // End of file indicator.
2430034 } FILE;
2430035 
2430036 extern FILE _stream[];       // Defined inside 'lib/stdio/FILE.c'.
2430037 
2430038 #define stdin   (&_stream[0])
2430039 #define stdout  (&_stream[1])
2430040 #define stderr  (&_stream[2])
2430041 //----------------------------------------------------------------------
2430042 void     clearerr   (FILE *fp);
2430043 int      fclose     (FILE *fp);
2430044 int      feof       (FILE *fp);
2430045 int      ferror     (FILE *fp);
2430046 int      fflush     (FILE *fp);
2430047 int      fgetc      (FILE *fp);
2430048 int      fgetpos    (FILE *restrict fp, fpos_t *restrict pos);
2430049 char    *fgets      (char *restrict string, int n, FILE *restrict fp);
2430050 int      fileno     (FILE *fp);
2430051 FILE    *fopen      (const char *path, const char *mode);
2430052 int      fprintf    (FILE *fp, char *restrict format, ...);
2430053 int      fputc      (int c, FILE *fp);
2430054 int      fputs      (const char *restrict string, FILE *restrict fp);
2430055 size_t   fread      (void *restrict buffer, size_t size, size_t nmemb,
2430056                      FILE *restrict fp);
```

```
2430057 FILE    *freopen    (const char *restrict path,
2430058                      const char *restrict mode,
2430059                      FILE *restrict fp);
2430060 int      fscanf     (FILE *restrict fp, const char *restrict format,
2430061                      ...);
2430062 int      fseek      (FILE *fp, long int offset, int whence);
2430063 int      fsetpos    (FILE *fp, const fpos_t *pos);
2430064 long int ftell      (FILE *fp);
2430065 off_t    ftello     (FILE *fp);
2430066 size_t   fwrite     (const void *restrict buffer, size_t size,
2430067                      size_t nmemb, FILE *restrict fp);
2430068 #define  getc(p)    (fgetc (p))
2430069 int      getchar    (void);
2430070 char    *gets       (char *string);
2430071 void     perror     (const char *string);
2430072 int      printf     (const char *restrict format, ...);
2430073 #define  putc(c, p) (fputc ((c), (p)))
2430074 #define  putchar(c) (fputc ((c), (stdout)))
2430075 int      puts       (const char *string);
2430076 void     rewind     (FILE *fp);
2430077 int      scanf      (const char *restrict format, ...);
2430078 void     setbuf     (FILE *restrict fp, char *restrict buffer);
2430079 int      setvbuf    (FILE *restrict fp, char *restrict buffer,
2430080                      int buf_mode, size_t size);
2430081 int      snprintf   (char *restrict string, size_t size,
2430082                      const char *restrict format, ...);
2430083 int      sprintf    (char *restrict string, const char *restrict format,
2430084                      ...);
2430085 int      sscanf     (char *restrict string, const char *restrict format,
2430086                      ...);
2430087 int      vfprintf   (FILE *fp, char *restrict format, va_list arg);
2430088 int      vfscanf    (FILE *restrict fp, const char *restrict format,
2430089                      va_list arg);
2430090 int      vprintf    (const char *restrict format, va_list arg);
2430091 int      vscanf     (const char *restrict format, va_list ap);
2430092 int      vsnprintf  (char *restrict string, size_t size,
2430093                      const char *restrict format, va_list arg);
2430094 int      vsprintf   (char *restrict string, const char *restrict format,
2430095                      va_list arg);
2430096 int      vsscanf    (const char *string, const char *format,
2430097                      va_list ap);
2430098
2430099 #endif
```

101.9.1 lib/stdio/FILE.c

«

Si veda la sezione 97.2.

```
2440001 #include <stdio.h>
2440002 //
2440003 // There must be room for at least 'FOPEN_MAX' elements.
2440004 //
2440005 FILE _stream[FOPEN_MAX];
2440006 //----------------------------------------------------------------------
2440007 void
2440008 _stdio_stream_setup (void)
2440009 {
2440010     _stream[0].fdn   = 0;
2440011     _stream[0].error = 0;
2440012     _stream[0].eof   = 0;
2440013 
2440014     _stream[1].fdn   = 1;
2440015     _stream[1].error = 0;
2440016     _stream[1].eof   = 0;
2440017 
2440018     _stream[2].fdn   = 2;
2440019     _stream[2].error = 0;
2440020     _stream[2].eof   = 0;
2440021 }
```

101.9.2 lib/stdio/clearerr.c

«

Si veda la sezione 94.9.

```
2450001 #include <stdio.h>
2450002 //----------------------------------------------------------------------
2450003 void
2450004 clearerr (FILE *fp)
2450005 {
2450006     if (fp != NULL)
2450007       {
2450008         fp->error = 0;
2450009         fp->eof   = 0;
2450010       }
2450011 }
```

101.9.3 lib/stdio/fclose.c

Si veda la sezione 94.27.

```
2460001 #include <stdio.h>
2460002 #include <unistd.h>
2460003 //----------------------------------------------------------------------
2460004 int
2460005 fclose (FILE *fp)
2460006 {
2460007     return (close (fp->fdn));
2460008 }
```

101.9.4 lib/stdio/feof.c

Si veda la sezione 94.28.

```
2470001 #include <stdio.h>
2470002 //----------------------------------------------------------------------
2470003 int
2470004 feof (FILE *fp)
2470005 {
2470006     if (fp != NULL)
2470007       {
2470008         return (fp->eof);
2470009       }
2470010     return (0);
2470011 }
```

101.9.5 lib/stdio/ferror.c

Si veda la sezione 94.29.

```
2480001 #include <stdio.h>
2480002 //----------------------------------------------------------------------
2480003 int
2480004 ferror (FILE *fp)
2480005 {
2480006     if (fp != NULL)
2480007       {
2480008         return (fp->error);
2480009       }
2480010     return (0);
2480011 }
```

101.9.6 lib/stdio/fflush.c

«

Si veda la sezione 94.30.

```
2490001 #include <stdio.h>
2490002 //----------------------------------------------------------------------
2490003 int
2490004 fflush (FILE *fp)
2490005 {
2490006     //
2490007     // The os16 library does not have any buffered data.
2490008     //
2490009     return (0);
2490010 }
```

101.9.7 lib/stdio/fgetc.c

«

Si veda la sezione 94.31.

```
2500001 #include <stdio.h>
2500002 #include <sys/types.h>
2500003 #include <unistd.h>
2500004 //----------------------------------------------------------------------
2500005 int
2500006 fgetc (FILE *fp)
2500007 {
2500008     ssize_t size_read;
2500009     int     c;          // Character read.
2500010     //
2500011     for (c = 0;;)
2500012       {
2500013         size_read = read (fp->fdn, &c, (size_t) 1);
2500014         //
2500015         if (size_read <= 0)
2500016           {
2500017             //
2500018             // It is the end of file (zero) otherwise there is a
2500019             // problem (a negative value): return 'EOF'.
2500020             //
2500021             return (EOF);
2500022           }
2500023         //
2500024         // Valid read: end of scan.
2500025         //
2500026         return (c);
2500027       }
2500028 }
```

101.9.8 lib/stdio/fgetpos.c

Si veda la sezione 94.32.

```
2510001 #include <stdio.h>
2510002 //----------------------------------------------------------------------
2510003 int
2510004 fgetpos (FILE *restrict fp, fpos_t *restrict pos)
2510005 {
2510006     long int position;
2510007     //
2510008     if (fp != NULL)
2510009       {
2510010         position = ftell (fp);
2510011         if (position >= 0)
2510012           {
2510013             *pos = position;
2510014             return (0);
2510015           }
2510016       }
2510017     return (-1);
2510018 }
```

101.9.9 lib/stdio/fgets.c

Si veda la sezione 94.33.

```
2520001 #include <stdio.h>
2520002 #include <sys/types.h>
2520003 #include <unistd.h>
2520004 #include <stddef.h>
2520005 //----------------------------------------------------------------------
2520006 char *
2520007 fgets (char *restrict string, int n, FILE *restrict fp)
2520008 {
2520009     ssize_t size_read;
2520010     int     b;          // Index inside the string buffer.
2520011     //
2520012     for (b = 0; b < (n-1); b++, string[b] = 0)
2520013       {
2520014         size_read = read (fp->fdn, &string[b], (size_t) 1);
2520015         //
2520016         if (size_read <= 0)
2520017           {
2520018             //
2520019             // It is the end of file (zero) otherwise there is a
2520020             // problem (a negative value).
2520021             //
2520022             string[b] = 0;
```

```
2520023                 break;
2520024               }
2520025             //
2520026             if (string[b] == '\n')
2520027               {
2520028                 b++;
2520029                 string[b] = 0;
2520030                 break;
2520031               }
2520032           }
2520033         //
2520034         // If `b` is zero, nothing was read and `NULL' is returned.
2520035         //
2520036         if (b == 0)
2520037           {
2520038             return (NULL);
2520039           }
2520040         else
2520041           {
2520042             return (string);
2520043           }
2520044     }
```

101.9.10 lib/stdio/fileno.c

Si veda la sezione 94.34.

```
2530001     #include <stdio.h>
2530002     #include <errno.h>
2530003     //----------------------------------------------------------------------
2530004     int
2530005     fileno (FILE *fp)
2530006     {
2530007         if (fp != NULL)
2530008           {
2530009             return (fp->fdn);
2530010           }
2530011         errset (EBADF);                    // Bad file descriptor.
2530012         return (-1);
2530013     }
```

101.9.11 lib/stdio/fopen.c

Si veda la sezione 94.35.

```
2540001 #include <fcntl.h>
2540002 #include <stdarg.h>
2540003 #include <stddef.h>
2540004 #include <string.h>
2540005 #include <errno.h>
2540006 #include <sys/os16.h>
2540007 #include <const.h>
2540008 #include <limits.h>
2540009 #include <stdio.h>
2540010
2540011 //----------------------------------------------------------------------
2540012 FILE *
2540013 fopen (const char *path, const char *mode)
2540014 {
2540015     int fdn;
2540016     //
2540017     if      (strcmp (mode, "r")   ||
2540018               strcmp (mode, "rb"))
2540019       {
2540020         fdn = open (path, O_RDONLY);
2540021       }
2540022     else if (strcmp (mode, "r+")  ||
2540023               strcmp (mode, "r+b") ||
2540024               strcmp (mode, "rb+"))
2540025       {
2540026         fdn = open (path, O_RDWR);
2540027       }
2540028     else if (strcmp (mode, "w")  ||
2540029               strcmp (mode, "wb"))
2540030       {
2540031         fdn = open (path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
2540032       }
2540033     else if (strcmp (mode, "w+")  ||
2540034               strcmp (mode, "w+b") ||
2540035               strcmp (mode, "wb+"))
2540036       {
2540037         fdn = open (path, O_RDWR|O_CREAT|O_TRUNC, 0666);
2540038       }
2540039     else if (strcmp (mode, "a")  ||
2540040               strcmp (mode, "ab"))
2540041       {
2540042         fdn = open (path, O_WRONLY|O_APPEND|O_CREAT|O_TRUNC, 0666);
2540043       }
2540044     else if (strcmp (mode, "a+")  ||
2540045               strcmp (mode, "a+b") ||
```

```
2540046                 strcmp (mode, "ab+"))
2540047           {
2540048             fdn = open (path, O_RDWR|O_APPEND|O_CREAT|O_TRUNC, 0666);
2540049           }
2540050         else
2540051           {
2540052             errset (EINVAL);                // Invalid argument.
2540053             return (NULL);
2540054           }
2540055         //
2540056         // Check the file descriptor returned.
2540057         //
2540058         if (fdn < 0)
2540059           {
2540060             //
2540061             // The variable 'errno' is already set.
2540062             //
2540063             errset (errno);
2540064             return (NULL);
2540065           }
2540066         //
2540067         // A valid file descriptor is available: convert it into a file
2540068         // stream. Please note that the file descriptor number must be
2540069         // saved inside the corresponding '_stream[]' array, because the
2540070         // file pointer do not have knowledge of the relative position
2540071         // inside the array.
2540072         //
2540073         _stream[fdn].fdn = fdn;      // Saved the file descriptor number.
2540074         //
2540075         return (&_stream[fdn]);      // Returned the file stream pointer.
2540076     }
```

101.9.12 lib/stdio/fprintf.c

Si veda la sezione 94.78.

```
2550001 #include <stdio.h>
2550002 
2550003 //----------------------------------------------------------------------
2550004 int
2550005 fprintf (FILE *fp, char *restrict format, ...)
2550006 {
2550007     va_list ap;
2550008     va_start (ap, format);
2550009     return (vfprintf (fp, format, ap));
2550010 }
```

101.9.13 lib/stdio/fputc.c

Si veda la sezione 94.37.

```
2560001 #include <stdio.h>
2560002 #include <sys/types.h>
2560003 #include <sys/os16.h>
2560004 #include <string.h>
2560005 #include <unistd.h>
2560006 //----------------------------------------------------------------------
2560007 int
2560008 fputc (int c, FILE *fp)
2560009 {
2560010     ssize_t size_written;
2560011     char    character = (char) c;
2560012     size_written = write (fp->fdn, &character, (size_t) 1);
2560013     if (size_written < 0)
2560014       {
2560015         fp->eof = 1;
2560016         return (EOF);
2560017       }
2560018     return (c);
2560019 }
```

101.9.14 lib/stdio/fputs.c

Si veda la sezione 94.38.

```
2570001 #include <stdio.h>
2570002 #include <string.h>
2570003 //----------------------------------------------------------------------
2570004 int
2570005 fputs (const char *restrict string, FILE *restrict fp)
2570006 {
2570007     int     i; // Index inside the string to be printed.
2570008     int status;
2570009 
2570010     for (i = 0; i < strlen (string); i++)
2570011       {
2570012         status = fputc (string[i], fp);
2570013         if (status == EOF)
2570014           {
2570015             fp->eof = 1;
2570016             return (EOF);
2570017           }
2570018       }
2570019     return (0);
2570020 }
```

101.9.15 lib/stdio/fread.c

Si veda la sezione 94.39.

```
2580001 #include <unistd.h>
2580002 #include <stdio.h>
2580003 //----------------------------------------------------------------------
2580004 size_t
2580005 fread (void *restrict buffer, size_t size, size_t nmemb,
2580006        FILE *restrict fp)
2580007 {
2580008     ssize_t size_read;
2580009     size_read = read (fp->fdn, buffer, (size_t) (size * nmemb));
2580010     if (size_read == 0)
2580011       {
2580012         fp->eof = 1;
2580013         return ((size_t) 0);
2580014       }
2580015     else if (size_read < 0)
2580016       {
2580017         fp->error = 1;
2580018         return ((size_t) 0);
2580019       }
2580020     else
2580021       {
2580022         return ((size_t) (size_read / size));
2580023       }
2580024 }
```

101.9.16 lib/stdio/freopen.c

Si veda la sezione 94.35.

```
2590001 #include <fcntl.h>
2590002 #include <stdarg.h>
2590003 #include <stddef.h>
2590004 #include <string.h>
2590005 #include <errno.h>
2590006 #include <sys/os16.h>
2590007 #include <const.h>
2590008 #include <limits.h>
2590009 #include <stdio.h>
2590010 
2590011 //----------------------------------------------------------------------
2590012 FILE *
2590013 freopen (const char *restrict path, const char *restrict mode,
2590014          FILE *restrict fp)
2590015 {
2590016     int   status;
```

```
2590017     FILE *fp_new;
2590018     //
2590019     if (fp == NULL)
2590020       {
2590021         return (NULL);
2590022       }
2590023     //
2590024     status = fclose (fp);
2590025     if (status != 0)
2590026       {
2590027         fp->error = 1;
2590028         return (NULL);
2590029       }
2590030     //
2590031     fp_new = fopen (path, mode);
2590032     //
2590033     if (fp_new == NULL)
2590034       {
2590035         return (NULL);
2590036       }
2590037     //
2590038     if (fp_new != fp)
2590039       {
2590040         fclose (fp_new);
2590041         return (NULL);
2590042       }
2590043     //
2590044     return (fp_new);
2590045 }
```

101.9.17 lib/stdio/fscanf.c

Si veda la sezione 94.90.

```
2600001 #include <stdio.h>
2600002 //----------------------------------------------------------------------
2600003 int
2600004 fscanf (FILE *restrict fp, const char *restrict format, ...)
2600005 {
2600006     va_list ap;
2600007     va_start (ap, format);
2600008     return vfscanf (fp, format, ap);
2600009 }
```

101.9.18 lib/stdio/fseek.c

«

Si veda la sezione 94.43.

```
2610001 #include <stdio.h>
2610002 #include <unistd.h>
2610003 //----------------------------------------------------------------------
2610004 int
2610005 fseek (FILE *fp, long int offset, int whence)
2610006 {
2610007     off_t off_new;
2610008     off_new = lseek (fp->fdn, (off_t) offset, whence);
2610009     if (off_new < 0)
2610010       {
2610011         fp->error = 1;
2610012         return (-1);
2610013       }
2610014     else
2610015       {
2610016         fp->eof = 0;
2610017         return (0);
2610018       }
2610019 }
```

101.9.19 lib/stdio/fseeko.c

«

Si veda la sezione 94.43.

```
2620001 #include <stdio.h>
2620002 #include <unistd.h>
2620003 //----------------------------------------------------------------------
2620004 int
2620005 fseeko (FILE *fp, off_t offset, int whence)
2620006 {
2620007     off_t off_new;
2620008     off_new = lseek (fp->fdn, offset, whence);
2620009     if (off_new < 0)
2620010       {
2620011         fp->error = 1;
2620012         return (-1);
2620013       }
2620014     else
2620015       {
2620016         return (0);
2620017       }
2620018 }
```

101.9.20 lib/stdio/fsetpos.c

Si veda la sezione 94.32.

```
2630001 #include <stdio.h>
2630002 //----------------------------------------------------------------------
2630003 int
2630004 fsetpos (FILE *restrict fp, fpos_t *restrict pos)
2630005 {
2630006     long int position;
2630007     //
2630008     if (fp != NULL)
2630009       {
2630010         position = fseek (fp, (long int) *pos, SEEK_SET);
2630011         if (position >= 0)
2630012           {
2630013             *pos = position;
2630014             return (0);
2630015           }
2630016       }
2630017     return (-1);
2630018 }
```

101.9.21 lib/stdio/ftell.c

Si veda la sezione 94.46.

```
2640001 #include <stdio.h>
2640002 #include <unistd.h>
2640003 //----------------------------------------------------------------------
2640004 long int
2640005 ftell (FILE *fp)
2640006 {
2640007     return ((long int) lseek (fp->fdn, (off_t) 0, SEEK_CUR));
2640008 }
```

101.9.22 lib/stdio/ftello.c

Si veda la sezione 94.46.

```
2650001 #include <stdio.h>
2650002 #include <unistd.h>
2650003 //----------------------------------------------------------------------
2650004 off_t
2650005 ftello (FILE *fp)
2650006 {
2650007     return (lseek (fp->fdn, (off_t) 0, SEEK_CUR));
2650008 }
```

101.9.23 lib/stdio/fwrite.c

«

Si veda la sezione 94.48.

```
2660001 #include <unistd.h>
2660002 #include <stdio.h>
2660003 //----------------------------------------------------------------------
2660004 size_t
2660005 fwrite (const void *restrict buffer, size_t size, size_t nmemb,
2660006         FILE *restrict fp)
2660007 {
2660008     ssize_t size_written;
2660009     size_written = write (fp->fdn, buffer, (size_t) (size * nmemb));
2660010     if (size_written < 0)
2660011       {
2660012         fp->error = 1;
2660013         return ((size_t) 0);
2660014       }
2660015     else
2660016       {
2660017         return ((size_t) (size_written / size));
2660018       }
2660019 }
```

101.9.24 lib/stdio/getchar.c

«

Si veda la sezione 94.31.

```
2670001 #include <stdio.h>
2670002 #include <sys/types.h>
2670003 #include <unistd.h>
2670004 //----------------------------------------------------------------------
2670005 int
2670006 getchar (void)
2670007 {
2670008     ssize_t size_read;
2670009     int     c;          // Character read.
2670010     //
2670011     for (c = 0;;)
2670012       {
2670013         size_read = read (STDIN_FILENO, &c, (size_t) 1);
2670014         //
2670015         if (size_read <= 0)
2670016           {
2670017             //
2670018             // It is the end of file (zero) otherwise there is a
2670019             // problem (a negative value): return 'EOF'.
2670020             //
2670021             _stream[STDIN_FILENO].eof = 1;
```

```
2670022                return (EOF);
2670023              }
2670024            //
2670025            // Valid read.
2670026            //
2670027            if (size_read == 0)
2670028              {
2670029                //
2670030                // If no character is ready inside the keyboard buffer, just
2670031                // retry.
2670032                //
2670033                continue;
2670034              }
2670035            //
2670036            // End of scan.
2670037            //
2670038            return (c);
2670039          }
2670040  }
```

101.9.25 lib/stdio/gets.c

«

Si veda la sezione 94.33.

```
2680001  #include <stdio.h>
2680002  #include <sys/types.h>
2680003  #include <unistd.h>
2680004  #include <stddef.h>
2680005  //----------------------------------------------------------------------
2680006  char *
2680007  gets (char *string)
2680008  {
2680009      ssize_t size_read;
2680010      int     b;           // Index inside the string buffer.
2680011      //
2680012      for (b = 0;; b++, string[b] = 0)
2680013        {
2680014          size_read = read (STDIN_FILENO, &string[b], (size_t) 1);
2680015          //
2680016          if (size_read <= 0)
2680017            {
2680018              //
2680019              // It is the end of file (zero) otherwise there is a
2680020              // problem (a negative value).
2680021              //
2680022              _stream[STDIN_FILENO].eof = 1;
2680023              string[b] = 0;
2680024              break;
```

```
2680025             }
2680026           //
2680027           if (string[b] == '\n')
2680028             {
2680029               b++;
2680030               string[b] = 0;
2680031               break;
2680032             }
2680033         }
2680034       //
2680035       // If `b` is zero, nothing was read and `NULL' is returned.
2680036       //
2680037       if (b == 0)
2680038         {
2680039           return (NULL);
2680040         }
2680041       else
2680042         {
2680043           return (string);
2680044         }
2680045 }
```

101.9.26 lib/stdio/perror.c

Si veda la sezione 94.77.

```
2690001 #include <stdio.h>
2690002 #include <errno.h>
2690003 #include <stddef.h>
2690004 #include <string.h>
2690005 //----------------------------------------------------------------------
2690006 void
2690007 perror (const char *string)
2690008 {
2690009     //
2690010     // If errno is zero, there is nothing to show.
2690011     //
2690012     if (errno == 0)
2690013       {
2690014         return;
2690015       }
2690016     //
2690017     // Show the string if there is one.
2690018     //
2690019     if (string != NULL && strlen (string) > 0)
2690020       {
2690021         printf ("%s: ", string);
2690022       }
```

```
2690023      //
2690024      // Show the translated error.
2690025      //
2690026      if (errfn[0] != 0 && errln != 0)
2690027        {
2690028          printf ("[%s:%u:%i] %s\n",
2690029                  errfn, errln, errno, strerror (errno));
2690030        }
2690031      else
2690032        {
2690033          printf ("[%i] %s\n", errno, strerror (errno));
2690034        }
2690035  }
```

101.9.27 lib/stdio/printf.c

Si veda la sezione 94.78.

```
2700001  #include <stdio.h>
2700002  //----------------------------------------------------------------------
2700003  int
2700004  printf (char *restrict format, ...)
2700005  {
2700006      va_list ap;
2700007      va_start (ap, format);
2700008      return (vprintf (format, ap));
2700009  }
```

101.9.28 lib/stdio/puts.c

Si veda la sezione 94.38.

```
2710001  #include <stdio.h>
2710002  //----------------------------------------------------------------------
2710003  int
2710004  puts (const char *string)
2710005  {
2710006      int status;
2710007      status = printf ("%s\n", string);
2710008      if (status < 0)
2710009        {
2710010          return (EOF);
2710011        }
2710012      else
2710013        {
2710014          return (status);
2710015        }
```

```
2710016 }
```

101.9.29 lib/stdio/rewind.c

«

Si veda la sezione 94.88.

```
2720001 #include <stdio.h>
2720002 //----------------------------------------------------------------------
2720003 void
2720004 rewind (FILE *fp)
2720005 {
2720006     (void) fseek (fp, 0L, SEEK_SET);
2720007     fp->error = 0;
2720008 }
```

101.9.30 lib/stdio/scanf.c

«

Si veda la sezione 94.90.

```
2730001 #include <stdio.h>
2730002 //----------------------------------------------------------------------
2730003 int
2730004 scanf (const char *restrict format, ...)
2730005 {
2730006     va_list ap;
2730007     va_start (ap, format);
2730008     return vfscanf (stdin, format, ap);
2730009 }
```

101.9.31 lib/stdio/setbuf.c

«

Si veda la sezione 94.93.

```
2740001 #include <stdio.h>
2740002 //----------------------------------------------------------------------
2740003 void
2740004 setbuf (FILE *restrict fp, char *restrict buffer)
2740005 {
2740006     //
2740007     // The os16 library does not have any buffered data.
2740008     //
2740009     return;
2740010 }
```

101.9.32 lib/stdio/setvbuf.c

Si veda la sezione 94.93.

```
2750001 #include <stdio.h>
2750002 //----------------------------------------------------------------------
2750003 int
2750004 setvbuf (FILE *restrict fp, char *restrict buffer, int buf_mode,
2750005          size_t size)
2750006 {
2750007     //
2750008     // The os16 library does not have any buffered data.
2750009     //
2750010     return (0);
2750011 }
```

101.9.33 lib/stdio/snprintf.c

Si veda la sezione 94.78.

```
2760001 #include <stdio.h>
2760002 #include <stdarg.h>
2760003 //----------------------------------------------------------------------
2760004 int
2760005 snprintf (char *restrict string, size_t size,
2760006           const char *restrict format, ...)
2760007 {
2760008     va_list ap;
2760009     va_start (ap, format);
2760010     return vsnprintf (string, size, format, ap);
2760011 }
```

101.9.34 lib/stdio/sprintf.c

Si veda la sezione 94.78.

```
2770001 #include <stdio.h>
2770002 #include <stdarg.h>
2770003 //----------------------------------------------------------------------
2770004 int
2770005 sprintf (char *restrict string, const char *restrict format,
2770006          ...)
2770007 {
2770008     va_list ap;
2770009     va_start (ap, format);
2770010     return vsnprintf (string, (size_t) BUFSIZ, format, ap);
2770011 }
```

101.9.35 lib/stdio/sscanf.c

«

Si veda la sezione 94.90.

```
2780001 #include <stdio.h>
2780002 //----------------------------------------------------------------------
2780003 int
2780004 sscanf (char *restrict string, const char *restrict format, ...)
2780005 {
2780006     va_list ap;
2780007     va_start (ap, format);
2780008     return vsscanf (string, format, ap);
2780009 }
```

101.9.36 lib/stdio/vfprintf.c

«

Si veda la sezione 94.128.

```
2790001 #include <stdio.h>
2790002 #include <sys/types.h>
2790003 #include <sys/os16.h>
2790004 #include <string.h>
2790005 #include <unistd.h>
2790006 //----------------------------------------------------------------------
2790007 int
2790008 vfprintf (FILE *fp, char *restrict format, va_list arg)
2790009 {
2790010     ssize_t     size_written;
2790011     size_t      size;
2790012     size_t      size_total;
2790013     int         status;
2790014     char        string[BUFSIZ];
2790015     char       *buffer = string;
2790016     //
2790017     buffer[0] = 0;
2790018     status    = vsprintf (buffer, format, arg);
2790019     //
2790020     size = strlen (buffer);
2790021     if (size >= BUFSIZ)
2790022       {
2790023         size = BUFSIZ;
2790024       }
2790025     //
2790026     for (size_total = 0, size_written = 0;
2790027          size_total < size;
2790028          size_total += size_written, buffer += size_written)
2790029       {
2790030         size_written = write (fp->fdn, buffer, size - size_total);
2790031         if (size_written < 0)
```

```
2790032             {
2790033               return (size_total);
2790034             }
2790035         }
2790036     return (size);
2790037 }
```

101.9.37 lib/stdio/vfscanf.c

Si veda la sezione 94.129.

```
2800001 #include <stdio.h>
2800002 
2800003 //----------------------------------------------------------------------
2800004 int vfsscanf (FILE *restrict fp, const char *string,
2800005               const char *restrict format, va_list ap);
2800006 //----------------------------------------------------------------------
2800007 int
2800008 vfscanf (FILE *restrict fp, const char *restrict format, va_list ap)
2800009 {
2800010     return (vfsscanf (fp, NULL, format, ap));
2800011 }
2800012 //----------------------------------------------------------------------
```

101.9.38 lib/stdio/vfsscanf.c

Si veda la sezione 94.129.

```
2810001 #include <stdint.h>
2810002 #include <stdbool.h>
2810003 #include <stdlib.h>
2810004 #include <string.h>
2810005 #include <stdio.h>
2810006 #include <stdarg.h>
2810007 #include <ctype.h>
2810008 #include <errno.h>
2810009 #include <stddef.h>
2810010 //----------------------------------------------------------------------
2810011 //
2810012 // This function is not standard and is able to do the work of both
2810013 // `vfscanf()' and `vsscanf()'.
2810014 //
2810015 //----------------------------------------------------------------------
2810016 #define WIDTH_MAX       64
2810017 //----------------------------------------------------------------------
2810018 static intmax_t strtointmax (const char *restrict string,
2810019                              char **restrict endptr, int base,
```

```
2810020                                  size_t max_width);
2810021 static int      ass_or_eof  (int consumed, int assigned);
2810022 //------------------------------------------------------------------------
2810023 int
2810024 vfsscanf (FILE *restrict fp, const char *string,
2810025           const char *restrict format, va_list ap)
2810026 {
2810027     int                  f               = 0;            // Format index.
2810028     char                 buffer[BUFSIZ];
2810029     const char          *input           = string;       // Default.
2810030     const char          *start           = input;        // Default.
2810031     char                *next            = NULL;
2810032     int                  scanned         = 0;
2810033     //
2810034     bool                 stream          = 0;
2810035     bool                 flag_star       = 0;
2810036     bool                 specifier       = 0;
2810037     bool                 specifier_flags = 0;
2810038     bool                 specifier_width = 0;
2810039     bool                 specifier_type  = 0;
2810040     bool                 inverted        = 0;
2810041     //
2810042     char                *ptr_char;
2810043     signed char         *ptr_schar;
2810044     unsigned char       *ptr_uchar;
2810045     short int           *ptr_sshort;
2810046     unsigned short int *ptr_ushort;
2810047     int                 *ptr_sint;
2810048     unsigned int        *ptr_uint;
2810049     long int            *ptr_slong;
2810050     unsigned long int   *ptr_ulong;
2810051     intmax_t            *ptr_simax;
2810052     uintmax_t           *ptr_uimax;
2810053     size_t              *ptr_size;
2810054     ptrdiff_t           *ptr_ptrdiff;
2810055     void               **ptr_void;
2810056     //
2810057     size_t               width;
2810058     char                 width_string[WIDTH_MAX+1];
2810059     int                  w;              // Index inside width string.
2810060     int                  assigned        = 0;    // Assignment counter.
2810061     int                  consumed        = 0;    // Consumed counter.
2810062     //
2810063     intmax_t             value_i;
2810064     uintmax_t            value_u;
2810065     //
2810066     const char          *end_format;
2810067     const char          *end_input;
```

```
2810068     int                   count;           // Generic counter.
2810069     int                   index;           // Generic index.
2810070     bool                  ascii[128];
2810071     //
2810072     void                 *pstatus;
2810073     //
2810074     // Initialize some data.
2810075     //
2810076     width_string[0] = '\0';
2810077     end_format      = format + (strlen (format));
2810078     //
2810079     // Check arguments and find where input comes.
2810080     //
2810081     if (fp == NULL && (string == NULL || string[0] == 0))
2810082       {
2810083         errset (EINVAL);                   // Invalid argument.
2810084         return (EOF);
2810085       }
2810086     //
2810087     if (fp != NULL && string != NULL && string[0] != 0)
2810088       {
2810089         errset (EINVAL);                   // Invalid argument.
2810090         return (EOF);
2810091       }
2810092     //
2810093     if (fp != NULL)
2810094       {
2810095         stream = 1;
2810096       }
2810097     //
2810098     //
2810099     //
2810100     for (;;)
2810101       {
2810102         if (stream)
2810103           {
2810104             pstatus = fgets (buffer, BUFSIZ, fp);
2810105             //
2810106             if (pstatus == NULL)
2810107               {
2810108                 return (ass_or_eof (consumed, assigned));
2810109               }
2810110             //
2810111             input = buffer;
2810112             start = input;
2810113             next  = NULL;
2810114           }
2810115         //
```

```
2810116           // Calculate end input.
2810117           //
2810118           end_input = input + (strlen (input));
2810119           //
2810120           // Scan format and input strings. Index 'f' is not reset.
2810121           //
2810122           while (&format[f] < end_format && input < end_input)
2810123             {
2810124               if (!specifier)
2810125                 {
2810126                   //--------------- The context is not inside a specifier.
2810127                   if (isspace (format[f]))
2810128                     {
2810129                       //------------------------------------------- Space.
2810130                       while (isspace (*input))
2810131                         {
2810132                           input++;
2810133                         }
2810134                       //
2810135                       // Verify that the input string is not finished.
2810136                       //
2810137                       if (input[0] == 0)
2810138                         {
2810139                           //
2810140                           // As the input string is finished, the format
2810141                           // string index is not advanced, because there
2810142                           // might be more spaces on the next line (if
2810143                           // there is a next line, of course).
2810144                           //
2810145                           continue;
2810146                         }
2810147                       else
2810148                         {
2810149                           f++;
2810150                           continue;
2810151                         }
2810152                     }
2810153                   if (format[f] != '%')
2810154                     {
2810155                       //------------------------------ Ordinary character.
2810156                       if (format[f] == *input)
2810157                         {
2810158                           input++;
2810159                           f++;
2810160                           continue;
2810161                         }
2810162                       else
2810163                         {
```

```
2810164                         return (ass_or_eof (consumed, assigned));
2810165                       }
2810166                   }
2810167                 if (format[f] == '%' && format[f+1] == '%')
2810168                   {
2810169                     //------------------------- Matching a literal '%'.
2810170                     f++;
2810171                     if (format[f] == *input)
2810172                       {
2810173                         input++;
2810174                         f++;
2810175                         continue;
2810176                       }
2810177                     else
2810178                       {
2810179                         return (ass_or_eof (consumed, assigned));
2810180                       }
2810181                   }
2810182                 if (format[f] == '%')
2810183                   {
2810184                     //------------------------- Percent of a specifier.
2810185                     f++;
2810186                     specifier       = 1;
2810187                     specifier_flags = 1;
2810188                     continue;
2810189                   }
2810190               }
2810191             //
2810192             if (specifier && specifier_flags)
2810193               {
2810194                 //--------------- The context is inside specifier flags.
2810195                 if (format[f] == '*')
2810196                   {
2810197                     //-------------------- Assignment suppression star.
2810198                     flag_star = 1;
2810199                     f++;
2810200                   }
2810201                 else
2810202                   {
2810203                     //----- End of flags and begin of specifier length.
2810204                     specifier_flags = 0;
2810205                     specifier_width = 1;
2810206                   }
2810207               }
2810208             //
2810209             if (specifier && specifier_width)
2810210               {
2810211                 //------------- The context is inside a specifier width.
```

```
2810212                     for (w = 0;
2810213                               format[f] >= '0'
2810214                            && format[f] <= '9'
2810215                            && w < WIDTH_MAX;
2810216                           w++)
2810217                       {
2810218                         width_string[w] = format[f];
2810219                         f++;
2810220                       }
2810221                     width_string[w] = '\0';
2810222                     width = atoi (width_string);
2810223                     if (width > WIDTH_MAX)
2810224                       {
2810225                         width = WIDTH_MAX;
2810226                       }
2810227                     //
2810228                     // A zero width means an unspecified limit for the field
2810229                     // length.
2810230                     //
2810231                     //--------- End of spec. width and begin of spec. type.
2810232                     specifier_width = 0;
2810233                     specifier_type  = 1;
2810234                   }
2810235                 //
2810236                 if (specifier && specifier_type)
2810237                   {
2810238                     //
2810239                     // Specifiers with length modifier.
2810240                     //
2810241                     if (format[f] == 'h' && format[f+1] == 'h')
2810242                       {
2810243                         //------------------------------------------- char.
2810244                         if      (format[f+2] == 'd')
2810245                           {
2810246                             //------------------------ signed char, base 10.
2810247                             value_i = strtointmax (input, &next, 10, width);
2810248                             if (input == next)
2810249                               {
2810250                                 return (ass_or_eof (consumed, assigned));
2810251                               }
2810252                             consumed++;
2810253                             if (!flag_star)
2810254                               {
2810255                                 ptr_schar = va_arg (ap, signed char *);
2810256                                *ptr_schar = value_i;
2810257                                 assigned++;
2810258                               }
2810259                             f += 3;
```

```
2810260                        input = next;
2810261                      }
2810262                    else if (format[f+2] == 'i')
2810263                      {
2810264                        //------------------- signed char, base unknown.
2810265                        value_i = strtointmax (input, &next, 0, width);
2810266                        if (input == next)
2810267                          {
2810268                            return (ass_or_eof (consumed, assigned));
2810269                          }
2810270                        consumed++;
2810271                        if (!flag_star)
2810272                          {
2810273                            ptr_schar = va_arg (ap, signed char *);
2810274                           *ptr_schar = value_i;
2810275                            assigned++;
2810276                          }
2810277                        f += 3;
2810278                        input = next;
2810279                      }
2810280                    else if (format[f+2] == 'o')
2810281                      {
2810282                        //------------------------- signed char, base 8.
2810283                        value_i = strtointmax (input, &next, 8, width);
2810284                        if (input == next)
2810285                          {
2810286                            return (ass_or_eof (consumed, assigned));
2810287                          }
2810288                        consumed++;
2810289                        if (!flag_star)
2810290                          {
2810291                            ptr_schar = va_arg (ap, signed char *);
2810292                           *ptr_schar = value_i;
2810293                            assigned++;
2810294                          }
2810295                        f += 3;
2810296                        input = next;
2810297                      }
2810298                    else if (format[f+2] == 'u')
2810299                      {
2810300                        //--------------------- unsigned char, base 10.
2810301                        value_u = strtointmax (input, &next, 10, width);
2810302                        if (input == next)
2810303                          {
2810304                            return (ass_or_eof (consumed, assigned));
2810305                          }
2810306                        consumed++;
2810307                        if (!flag_star)
```

```
2810308                               {
2810309                                 ptr_uchar = va_arg (ap, unsigned char *);
2810310                                *ptr_uchar = value_u;
2810311                                 assigned++;
2810312                               }
2810313                             f += 3;
2810314                             input = next;
2810315                           }
2810316                         else if (format[f+2] == 'x' || format[f+2] == 'X')
2810317                           {
2810318                             //----------------------- signed char, base 16.
2810319                             value_i = strtointmax (input, &next, 16, width);
2810320                             if (input == next)
2810321                               {
2810322                                 return (ass_or_eof (consumed, assigned));
2810323                               }
2810324                             consumed++;
2810325                             if (!flag_star)
2810326                               {
2810327                                 ptr_schar = va_arg (ap, signed char *);
2810328                                *ptr_schar = value_i;
2810329                                 assigned++;
2810330                               }
2810331                             f += 3;
2810332                             input = next;
2810333                           }
2810334                         else if (format[f+2] == 'n')
2810335                           {
2810336                             //---------- signed char, string index counter.
2810337                             ptr_schar = va_arg (ap, signed char *);
2810338                            *ptr_schar = (signed char)
2810339                                             (input - start + scanned);
2810340                             f += 3;
2810341                           }
2810342                         else
2810343                           {
2810344                             //----------- unsupported or unknown specifier.
2810345                             f += 2;
2810346                           }
2810347                       }
2810348                     else if (format[f] == 'h')
2810349                       {
2810350                         //------------------------------------------ short.
2810351                         if      (format[f+1] == 'd')
2810352                           {
2810353                             //---------------------- signed short, base 10.
2810354                             value_i = strtointmax (input, &next, 10, width);
2810355                             if (input == next)
```

```
2810356                             {
2810357                               return (ass_or_eof (consumed, assigned));
2810358                             }
2810359                           consumed++;
2810360                           if (!flag_star)
2810361                             {
2810362                               ptr_sshort = va_arg (ap, signed short *);
2810363                              *ptr_sshort = value_i;
2810364                               assigned++;
2810365                             }
2810366                           f += 2;
2810367                           input = next;
2810368                         }
2810369                       else if (format[f+1] == 'i')
2810370                         {
2810371                           //------------------ signed short, base unknown.
2810372                           value_i = strtointmax (input, &next, 0, width);
2810373                           if (input == next)
2810374                             {
2810375                               return (ass_or_eof (consumed, assigned));
2810376                             }
2810377                           consumed++;
2810378                           if (!flag_star)
2810379                             {
2810380                               ptr_sshort = va_arg (ap, signed short *);
2810381                              *ptr_sshort = value_i;
2810382                               assigned++;
2810383                             }
2810384                           f += 2;
2810385                           input = next;
2810386                         }
2810387                       else if (format[f+1] == 'o')
2810388                         {
2810389                           //------------------------ signed short, base 8.
2810390                           value_i = strtointmax (input, &next, 8, width);
2810391                           if (input == next)
2810392                             {
2810393                               return (ass_or_eof (consumed, assigned));
2810394                             }
2810395                           consumed++;
2810396                           if (!flag_star)
2810397                             {
2810398                               ptr_sshort = va_arg (ap, signed short *);
2810399                              *ptr_sshort = value_i;
2810400                               assigned++;
2810401                             }
2810402                           f += 2;
2810403                           input = next;
```

```
2810404                     }
2810405                   else if (format[f+1] == 'u')
2810406                     {
2810407                       //-------------------- unsigned short, base 10.
2810408                       value_u = strtointmax (input, &next, 10, width);
2810409                       if (input == next)
2810410                         {
2810411                           return (ass_or_eof (consumed, assigned));
2810412                         }
2810413                       consumed++;
2810414                       if (!flag_star)
2810415                         {
2810416                           ptr_ushort = va_arg (ap, unsigned short *);
2810417                          *ptr_ushort = value_u;
2810418                           assigned++;
2810419                         }
2810420                       f += 2;
2810421                       input = next;
2810422                     }
2810423                   else if (format[f+1] == 'x' || format[f+2] == 'X')
2810424                     {
2810425                       //---------------------- signed short, base 16.
2810426                       value_i = strtointmax (input, &next, 16, width);
2810427                       if (input == next)
2810428                         {
2810429                           return (ass_or_eof (consumed, assigned));
2810430                         }
2810431                       consumed++;
2810432                       if (!flag_star)
2810433                         {
2810434                           ptr_sshort = va_arg (ap, signed short *);
2810435                          *ptr_sshort = value_i;
2810436                           assigned++;
2810437                         }
2810438                       f += 2;
2810439                       input = next;
2810440                     }
2810441                   else if (format[f+1] == 'n')
2810442                     {
2810443                       //----------- signed char, string index counter.
2810444                       ptr_sshort = va_arg (ap, signed short *);
2810445                      *ptr_sshort = (signed short)
2810446                                     (input - start + scanned);
2810447                       f += 2;
2810448                     }
2810449                   else
2810450                     {
2810451                       //------------ unsupported or unknown specifier.
```

```
2810452                         f += 1;
2810453                       }
2810454                   }
2810455                 //------------------------ There is no ‘long long int’.
2810456                 else if (format[f] == 'l')
2810457                   {
2810458                     //--------------------------------------- long int.
2810459                     if      (format[f+1] == 'd')
2810460                       {
2810461                         //---------------------- signed long, base 10.
2810462                         value_i = strtointmax (input, &next, 10, width);
2810463                         if (input == next)
2810464                           {
2810465                             return (ass_or_eof (consumed, assigned));
2810466                           }
2810467                         consumed++;
2810468                         if (!flag_star)
2810469                           {
2810470                             ptr_slong = va_arg (ap, signed long *);
2810471                            *ptr_slong = value_i;
2810472                             assigned++;
2810473                           }
2810474                         f += 2;
2810475                         input = next;
2810476                       }
2810477                     else if (format[f+1] == 'i')
2810478                       {
2810479                         //----------------- signed long, base unknown.
2810480                         value_i = strtointmax (input, &next, 0, width);
2810481                         if (input == next)
2810482                           {
2810483                             return (ass_or_eof (consumed, assigned));
2810484                           }
2810485                         consumed++;
2810486                         if (!flag_star)
2810487                           {
2810488                             ptr_slong = va_arg (ap, signed long *);
2810489                            *ptr_slong = value_i;
2810490                             assigned++;
2810491                           }
2810492                         f += 2;
2810493                         input = next;
2810494                       }
2810495                     else if (format[f+1] == 'o')
2810496                       {
2810497                         //----------------------- signed long, base 8.
2810498                         value_i = strtointmax (input, &next, 8, width);
2810499                         if (input == next)
```

```
2810500                                 {
2810501                                   return (ass_or_eof (consumed, assigned));
2810502                                 }
2810503                               consumed++;
2810504                               if (!flag_star)
2810505                                 {
2810506                                   ptr_slong = va_arg (ap, signed long *);
2810507                                  *ptr_slong = value_i;
2810508                                   assigned++;
2810509                                 }
2810510                               f += 2;
2810511                               input = next;
2810512                             }
2810513                           else if (format[f+1] == 'u')
2810514                             {
2810515                               //--------------------- unsigned long, base 10.
2810516                               value_u = strtointmax (input, &next, 10, width);
2810517                               if (input == next)
2810518                                 {
2810519                                   return (ass_or_eof (consumed, assigned));
2810520                                 }
2810521                               consumed++;
2810522                               if (!flag_star)
2810523                                 {
2810524                                   ptr_ulong = va_arg (ap, unsigned long *);
2810525                                  *ptr_ulong = value_u;
2810526                                   assigned++;
2810527                                 }
2810528                               f += 2;
2810529                               input = next;
2810530                             }
2810531                           else if (format[f+1] == 'x' || format[f+2] == 'X')
2810532                             {
2810533                               //----------------------- signed long, base 16.
2810534                               value_i = strtointmax (input, &next, 16, width);
2810535                               if (input == next)
2810536                                 {
2810537                                   return (ass_or_eof (consumed, assigned));
2810538                                 }
2810539                               consumed++;
2810540                               if (!flag_star)
2810541                                 {
2810542                                   ptr_slong = va_arg (ap, signed long *);
2810543                                  *ptr_slong = value_i;
2810544                                   assigned++;
2810545                                 }
2810546                               f += 2;
2810547                               input = next;
```

```
2810548                             }
2810549                           else if (format[f+1] == 'n')
2810550                             {
2810551                               //----------- signed char, string index counter.
2810552                               ptr_slong = va_arg (ap, signed long *);
2810553                              *ptr_slong = (signed long)
2810554                                           (input - start + scanned);
2810555                               f += 2;
2810556                             }
2810557                           else
2810558                             {
2810559                               //------------ unsupported or unknown specifier.
2810560                               f += 1;
2810561                             }
2810562                         }
2810563                       else if (format[f] == 'j')
2810564                         {
2810565                           //---------------------------------------- intmax_t.
2810566                           if      (format[f+1] == 'd')
2810567                             {
2810568                               //-------------------------- intmax_t, base 10.
2810569                               value_i = strtointmax (input, &next, 10, width);
2810570                               if (input == next)
2810571                                 {
2810572                                   return (ass_or_eof (consumed, assigned));
2810573                                 }
2810574                               consumed++;
2810575                               if (!flag_star)
2810576                                 {
2810577                                   ptr_simax = va_arg (ap, intmax_t *);
2810578                                  *ptr_simax = value_i;
2810579                                   assigned++;
2810580                                 }
2810581                               f += 2;
2810582                               input = next;
2810583                             }
2810584                           else if (format[f+1] == 'i')
2810585                             {
2810586                               //--------------------- intmax_t, base unknown.
2810587                               value_i = strtointmax (input, &next, 0, width);
2810588                               if (input == next)
2810589                                 {
2810590                                   return (ass_or_eof (consumed, assigned));
2810591                                 }
2810592                               consumed++;
2810593                               if (!flag_star)
2810594                                 {
2810595                                   ptr_simax = va_arg (ap, intmax_t *);
```

```
2810596                                *ptr_simax = value_i;
2810597                                 assigned++;
2810598                               }
2810599                             f += 2;
2810600                             input = next;
2810601                           }
2810602                         else if (format[f+1] == 'o')
2810603                           {
2810604                             //--------------------------- intmax_t, base 8.
2810605                             value_i = strtointmax (input, &next, 8, width);
2810606                             if (input == next)
2810607                               {
2810608                                 return (ass_or_eof (consumed, assigned));
2810609                               }
2810610                             consumed++;
2810611                             if (!flag_star)
2810612                               {
2810613                                 ptr_simax = va_arg (ap, intmax_t *);
2810614                                *ptr_simax = value_i;
2810615                                 assigned++;
2810616                               }
2810617                             f += 2;
2810618                             input = next;
2810619                           }
2810620                         else if (format[f+1] == 'u')
2810621                           {
2810622                             //------------------------- uintmax_t, base 10.
2810623                             value_u = strtointmax (input, &next, 10, width);
2810624                             if (input == next)
2810625                               {
2810626                                 return (ass_or_eof (consumed, assigned));
2810627                               }
2810628                             consumed++;
2810629                             if (!flag_star)
2810630                               {
2810631                                 ptr_uimax = va_arg (ap, uintmax_t *);
2810632                                *ptr_uimax = value_u;
2810633                                 assigned++;
2810634                               }
2810635                             f += 2;
2810636                             input = next;
2810637                           }
2810638                         else if (format[f+1] == 'x' || format[f+2] == 'X')
2810639                           {
2810640                             //-------------------------- intmax_t, base 16.
2810641                             value_i = strtointmax (input, &next, 16, width);
2810642                             if (input == next)
2810643                               {
```

```
2810644                                 return (ass_or_eof (consumed, assigned));
2810645                               }
2810646                             consumed++;
2810647                             if (!flag_star)
2810648                               {
2810649                                 ptr_simax = va_arg (ap, intmax_t *);
2810650                                *ptr_simax = value_i;
2810651                                 assigned++;
2810652                               }
2810653                             f += 2;
2810654                             input = next;
2810655                           }
2810656                         else if (format[f+1] == 'n')
2810657                           {
2810658                             //---------- signed char, string index counter.
2810659                             ptr_simax = va_arg (ap, intmax_t *);
2810660                            *ptr_simax = (intmax_t)
2810661                                         (input - start + scanned);
2810662                             f += 2;
2810663                           }
2810664                         else
2810665                           {
2810666                             //----------- unsupported or unknown specifier.
2810667                             f += 1;
2810668                           }
2810669                       }
2810670                     else if (format[f] == 'z')
2810671                       {
2810672                         //------------------------------------- size_t.
2810673                         if      (format[f+1] == 'd')
2810674                           {
2810675                             //--------------------------- size_t, base 10.
2810676                             value_i = strtointmax (input, &next, 10, width);
2810677                             if (input == next)
2810678                               {
2810679                                 return (ass_or_eof (consumed, assigned));
2810680                               }
2810681                             consumed++;
2810682                             if (!flag_star)
2810683                               {
2810684                                 ptr_size = va_arg (ap, size_t *);
2810685                                *ptr_size = value_i;
2810686                                 assigned++;
2810687                               }
2810688                             f += 2;
2810689                             input = next;
2810690                           }
2810691                         else if (format[f+1] == 'i')
```

```
2810692                  {
2810693                    //----------------------- size_t, base unknown.
2810694                    value_i = strtointmax (input, &next, 0, width);
2810695                    if (input == next)
2810696                      {
2810697                        return (ass_or_eof (consumed, assigned));
2810698                      }
2810699                    consumed++;
2810700                    if (!flag_star)
2810701                      {
2810702                        ptr_size = va_arg (ap, size_t *);
2810703                       *ptr_size = value_i;
2810704                        assigned++;
2810705                      }
2810706                    f += 2;
2810707                    input = next;
2810708                  }
2810709                else if (format[f+1] == 'o')
2810710                  {
2810711                    //---------------------------- size_t, base 8.
2810712                    value_i = strtointmax (input, &next, 8, width);
2810713                    if (input == next)
2810714                      {
2810715                        return (ass_or_eof (consumed, assigned));
2810716                      }
2810717                    consumed++;
2810718                    if (!flag_star)
2810719                      {
2810720                        ptr_size = va_arg (ap, size_t *);
2810721                       *ptr_size = value_i;
2810722                        assigned++;
2810723                      }
2810724                    f += 2;
2810725                    input = next;
2810726                  }
2810727                else if (format[f+1] == 'u')
2810728                  {
2810729                    //--------------------------- size_t, base 10.
2810730                    value_u = strtointmax (input, &next, 10, width);
2810731                    if (input == next)
2810732                      {
2810733                        return (ass_or_eof (consumed, assigned));
2810734                      }
2810735                    consumed++;
2810736                    if (!flag_star)
2810737                      {
2810738                        ptr_size = va_arg (ap, size_t *);
2810739                       *ptr_size = value_u;
```

```
2810740                                  assigned++;
2810741                                }
2810742                              f += 2;
2810743                              input = next;
2810744                            }
2810745                          else if (format[f+1] == 'x' || format[f+2] == 'X')
2810746                            {
2810747                              //---------------------------- size_t, base 16.
2810748                              value_i = strtointmax (input, &next, 16, width);
2810749                              if (input == next)
2810750                                {
2810751                                  return (ass_or_eof (consumed, assigned));
2810752                                }
2810753                              consumed++;
2810754                              if (!flag_star)
2810755                                {
2810756                                  ptr_size = va_arg (ap, size_t *);
2810757                                 *ptr_size = value_i;
2810758                                  assigned++;
2810759                                }
2810760                              f += 2;
2810761                              input = next;
2810762                            }
2810763                          else if (format[f+1] == 'n')
2810764                            {
2810765                              //----------- signed char, string index counter.
2810766                              ptr_size = va_arg (ap, size_t *);
2810767                             *ptr_size = (size_t) (input - start + scanned);
2810768                              f += 2;
2810769                            }
2810770                          else
2810771                            {
2810772                              //------------ unsupported or unknown specifier.
2810773                              f += 1;
2810774                            }
2810775                        }
2810776                      else if (format[f] == 't')
2810777                        {
2810778                          //---------------------------------------- ptrdiff_t.
2810779                          if      (format[f+1] == 'd')
2810780                            {
2810781                              //------------------------- ptrdiff_t, base 10.
2810782                              value_i = strtointmax (input, &next, 10, width);
2810783                              if (input == next)
2810784                                {
2810785                                  return (ass_or_eof (consumed, assigned));
2810786                                }
2810787                              consumed++;
```

```
2810788                         if (!flag_star)
2810789                           {
2810790                             ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810791                            *ptr_ptrdiff = value_i;
2810792                             assigned++;
2810793                           }
2810794                         f += 2;
2810795                         input = next;
2810796                       }
2810797                     else if (format[f+1] == 'i')
2810798                       {
2810799                         //--------------------- ptrdiff_t, base unknown.
2810800                         value_i = strtointmax (input, &next, 0, width);
2810801                         if (input == next)
2810802                           {
2810803                             return (ass_or_eof (consumed, assigned));
2810804                           }
2810805                         consumed++;
2810806                         if (!flag_star)
2810807                           {
2810808                             ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810809                            *ptr_ptrdiff = value_i;
2810810                             assigned++;
2810811                           }
2810812                         f += 2;
2810813                         input = next;
2810814                       }
2810815                     else if (format[f+1] == 'o')
2810816                       {
2810817                         //-------------------------- ptrdiff_t, base 8.
2810818                         value_i = strtointmax (input, &next, 8, width);
2810819                         if (input == next)
2810820                           {
2810821                             return (ass_or_eof (consumed, assigned));
2810822                           }
2810823                         consumed++;
2810824                         if (!flag_star)
2810825                           {
2810826                             ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810827                            *ptr_ptrdiff = value_i;
2810828                             assigned++;
2810829                           }
2810830                         f += 2;
2810831                         input = next;
2810832                       }
2810833                     else if (format[f+1] == 'u')
2810834                       {
2810835                         //------------------------- ptrdiff_t, base 10.
```

```
2810836                         value_u = strtointmax (input, &next, 10, width);
2810837                         if (input == next)
2810838                           {
2810839                             return (ass_or_eof (consumed, assigned));
2810840                           }
2810841                         consumed++;
2810842                         if (!flag_star)
2810843                           {
2810844                             ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810845                            *ptr_ptrdiff = value_u;
2810846                             assigned++;
2810847                           }
2810848                         f += 2;
2810849                         input = next;
2810850                       }
2810851                     else if (format[f+1] == 'x' || format[f+2] == 'X')
2810852                       {
2810853                         //------------------------- ptrdiff_t, base 16.
2810854                         value_i = strtointmax (input, &next, 16, width);
2810855                         if (input == next)
2810856                           {
2810857                             return (ass_or_eof (consumed, assigned));
2810858                           }
2810859                         consumed++;
2810860                         if (!flag_star)
2810861                           {
2810862                             ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810863                            *ptr_ptrdiff = value_i;
2810864                             assigned++;
2810865                           }
2810866                         f += 2;
2810867                         input = next;
2810868                       }
2810869                     else if (format[f+1] == 'n')
2810870                       {
2810871                         //----------- signed char, string index counter.
2810872                         ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810873                        *ptr_ptrdiff = (ptrdiff_t)
2810874                                       (input - start + scanned);
2810875                         f += 2;
2810876                       }
2810877                     else
2810878                       {
2810879                         //------------ unsupported or unknown specifier.
2810880                         f += 1;
2810881                       }
2810882                   }
2810883                 //
```

```
2810884                    // Specifiers with no length modifier.
2810885                    //
2810886                    if      (format[f] == 'd')
2810887                      {
2810888                        //----------------------- signed short, base 10.
2810889                        value_i = strtointmax (input, &next, 10, width);
2810890                        if (input == next)
2810891                          {
2810892                            return (ass_or_eof (consumed, assigned));
2810893                          }
2810894                        consumed++;
2810895                        if (!flag_star)
2810896                          {
2810897                            ptr_sshort = va_arg (ap, signed short *);
2810898                           *ptr_sshort = value_i;
2810899                            assigned++;
2810900                          }
2810901                        f += 1;
2810902                        input = next;
2810903                      }
2810904                    else if (format[f] == 'i')
2810905                      {
2810906                        //------------------------ signed int, base unknown.
2810907                        value_i = strtointmax (input, &next, 0, width);
2810908                        if (input == next)
2810909                          {
2810910                            return (ass_or_eof (consumed, assigned));
2810911                          }
2810912                        consumed++;
2810913                        if (!flag_star)
2810914                          {
2810915                            ptr_sint = va_arg (ap, signed int *);
2810916                           *ptr_sint = value_i;
2810917                            assigned++;
2810918                          }
2810919                        f += 1;
2810920                        input = next;
2810921                      }
2810922                    else if (format[f] == 'o')
2810923                      {
2810924                        //----------------------------- signed int, base 8.
2810925                        value_i = strtointmax (input, &next, 8, width);
2810926                        if (input == next)
2810927                          {
2810928                            return (ass_or_eof (consumed, assigned));
2810929                          }
2810930                        consumed++;
2810931                        if (!flag_star)
```

```
2810932                           {
2810933                             ptr_sint = va_arg (ap, signed int *);
2810934                            *ptr_sint = value_i;
2810935                             assigned++;
2810936                           }
2810937                         f += 1;
2810938                         input = next;
2810939                       }
2810940                     else if (format[f] == 'u')
2810941                       {
2810942                         //----------------------- unsigned short, base 10.
2810943                         value_u = strtointmax (input, &next, 10, width);
2810944                         if (input == next)
2810945                           {
2810946                             return (ass_or_eof (consumed, assigned));
2810947                           }
2810948                         consumed++;
2810949                         if (!flag_star)
2810950                           {
2810951                             ptr_uint = va_arg (ap, unsigned int *);
2810952                            *ptr_uint = value_u;
2810953                             assigned++;
2810954                           }
2810955                         f += 1;
2810956                         input = next;
2810957                       }
2810958                     else if (format[f] == 'x' || format[f] == 'X')
2810959                       {
2810960                         //------------------------- signed short, base 16.
2810961                         value_i = strtointmax (input, &next, 16, width);
2810962                         if (input == next)
2810963                           {
2810964                             return (ass_or_eof (consumed, assigned));
2810965                           }
2810966                         consumed++;
2810967                         if (!flag_star)
2810968                           {
2810969                             ptr_sint = va_arg (ap, signed int *);
2810970                            *ptr_sint = value_i;
2810971                             assigned++;
2810972                           }
2810973                         f += 1;
2810974                         input = next;
2810975                       }
2810976                     else if (format[f] == 'c')
2810977                       {
2810978                         //------------------------------------------ char[].
2810979                         if (width == 0) width = 1;
```

```
2810980                 //
2810981                 if (!flag_star) ptr_char = va_arg (ap, char *);
2810982                 //
2810983                 for (count = 0;
2810984                      width > 0 && *input != 0;
2810985                      width--, ptr_char++, input++)
2810986                   {
2810987                     if (!flag_star) *ptr_char = *input;
2810988                     //
2810989                     count++;
2810990                   }
2810991                 //
2810992                 if (count)                 consumed++;
2810993                 if (count && !flag_star) assigned++;
2810994                 //
2810995                 f += 1;
2810996               }
2810997             else if (format[f] == 's')
2810998               {
2810999                 //----------------------------------------- string.
2811000                 if (!flag_star) ptr_char = va_arg (ap, char *);
2811001                 //
2811002                 for (count = 0;
2811003                      !isspace (*input) && *input != 0;
2811004                      ptr_char++, input++)
2811005                   {
2811006                     if (!flag_star) *ptr_char = *input;
2811007                     //
2811008                     count++;
2811009                   }
2811010                 if (!flag_star) *ptr_char = 0;
2811011                 //
2811012                 if (count)                 consumed++;
2811013                 if (count && !flag_star) assigned++;
2811014                 //
2811015                 f += 1;
2811016               }
2811017             else if (format[f] == '[')
2811018               {
2811019                 //
2811020                 f++;
2811021                 //
2811022                 if (format[f] == '^')
2811023                   {
2811024                     inverted = 1;
2811025                     f++;
2811026                   }
2811027                 else
```

```
2811028                                {
2811029                                  inverted = 0;
2811030                                }
2811031                              //
2811032                              // Reset ascii array.
2811033                              //
2811034                              for (index = 0; index < 128; index++)
2811035                                {
2811036                                  ascii[index] = inverted;
2811037                                }
2811038                              //
2811039                              //
2811040                              //
2811041                              for (count = 0;&format[f] < end_format; count++)
2811042                                {
2811043                                  if (format[f] == ']' && count > 0)
2811044                                    {
2811045                                      break;
2811046                                    }
2811047                                  //
2811048                                  // Check for an interval.
2811049                                  //
2811050                                  if (format[f+1] == '-'
2811051                                   && format[f+2] != ']'
2811052                                   && format[f+2] != 0)
2811053                                    {
2811054                                      //
2811055                                      // Interval.
2811056                                      //
2811057                                      for (index = format[f];
2811058                                           index <= format[f+2];
2811059                                           index++)
2811060                                        {
2811061                                          ascii[index] = !inverted;
2811062                                        }
2811063                                      f += 3;
2811064                                      continue;
2811065                                    }
2811066                                  //
2811067                                  // Single character.
2811068                                  //
2811069                                  index = format[f];
2811070                                  ascii[index] = !inverted;
2811071                                  f++;
2811072                                }
2811073                              //
2811074                              // Is the scan correctly finished?.
2811075                              //
```

```
2811076                     if (format[f] != ']')
2811077                       {
2811078                         return (ass_or_eof (consumed, assigned));
2811079                       }
2811080                     //
2811081                     // The ascii table is populated.
2811082                     //
2811083                     if (width == 0) width = SIZE_MAX;
2811084                     //
2811085                     // Scan the input string.
2811086                     //
2811087                     if (!flag_star) ptr_char = va_arg (ap, char *);
2811088                     //
2811089                     for (count = 0;
2811090                          width > 0 && *input != 0;
2811091                          width--, ptr_char++, input++)
2811092                       {
2811093                         index = *input;
2811094                         if (ascii[index])
2811095                           {
2811096                             if (!flag_star) *ptr_char = *input;
2811097                             count++;
2811098                           }
2811099                         else
2811100                           {
2811101                             break;
2811102                           }
2811103                       }
2811104                     //
2811105                     if (count)               consumed++;
2811106                     if (count && !flag_star) assigned++;
2811107                     //
2811108                     f += 1;
2811109                   }
2811110                 else if (format[f] == 'p')
2811111                   {
2811112                     //----------------------------------------- void *.
2811113                     value_i = strtointmax (input, &next, 16, width);
2811114                     if (input == next)
2811115                       {
2811116                         return (ass_or_eof (consumed, assigned));
2811117                       }
2811118                     consumed++;
2811119                     if (!flag_star)
2811120                       {
2811121                         ptr_void = va_arg (ap, void **);
2811122                        *ptr_void = (void *) ((int) value_i);
2811123                         assigned++;
```

```
2811124                         }
2811125                       f += 1;
2811126                       input = next;
2811127                     }
2811128                   else if (format[f] == 'n')
2811129                     {
2811130                       //--------------- signed char, string index counter.
2811131                       ptr_sint = va_arg (ap, signed int *);
2811132                      *ptr_sint = (signed char) (input - start + scanned);
2811133                       f += 1;
2811134                     }
2811135                   else
2811136                     {
2811137                       //---------------- unsupported or unknown specifier.
2811138                       ;
2811139                     }
2811140 
2811141                   //--------------------------------------------------------
2811142                   // End of specifier.
2811143                   //--------------------------------------------------------
2811144 
2811145                   width_string[0]     = '\0';
2811146                   specifier           = 0;
2811147                   specifier_flags     = 0;
2811148                   specifier_width     = 0;
2811149                   specifier_type      = 0;
2811150                   flag_star           = 0;
2811151 
2811152                 }
2811153             }
2811154           //
2811155           // The format or the input string is terminated.
2811156           //
2811157           if  (&format[f] < end_format && stream)
2811158             {
2811159               //
2811160               // Only the input string is finished, and the input comes
2811161               // from a stream, so another read will be done.
2811162               //
2811163               scanned += (int) (input - start);
2811164               continue;
2811165             }
2811166           //
2811167           // The format string is terminated.
2811168           //
2811169           return (ass_or_eof (consumed, assigned));
2811170         }
2811171 }
```

```
2811172 //----------------------------------------------------------------------
2811173 static intmax_t
2811174 strtointmax (const char *restrict string, char **endptr,
2811175              int base, size_t max_width)
2811176 {
2811177     int      i;
2811178     int      d;                  // Digits counter.
2811179     int      sign = +1;
2811180     intmax_t number;
2811181     intmax_t previous;
2811182     int      digit;
2811183     //
2811184     bool     flag_prefix_oct = 0;
2811185     bool     flag_prefix_exa = 0;
2811186     bool     flag_prefix_dec = 0;
2811187     //
2811188     // If the 'max_width' value is zero, fix it to the maximum
2811189     // that it can represent.
2811190     //
2811191     if (max_width == 0)
2811192       {
2811193         max_width = SIZE_MAX;
2811194       }
2811195     //
2811196     // Eat initial spaces, but if there are spaces, there is an
2811197     // error inside the calling function!
2811198     //
2811199     for (i = 0; isspace (string[i]); i++)
2811200       {
2811201         fprintf (stderr, "libc error: file \"%s\", line %i\n",
2811202                  __FILE__, __LINE__);
2811203         ;
2811204       }
2811205     //
2811206     // Check sign. The 'max_width' counts also the sign, if there is
2811207     // one.
2811208     //
2811209     if (string[i] == '+')
2811210       {
2811211         sign = +1;
2811212         i++;
2811213         max_width--;
2811214       }
2811215     else if (string[i] == '-')
2811216       {
2811217         sign = -1;
2811218         i++;
2811219         max_width--;
```

```
2811220         }
2811221       //
2811222       // Check for prefix.
2811223       //
2811224       if (string[i] == '0')
2811225         {
2811226           if (string[i+1] == 'x' || string[i+1] == 'X')
2811227             {
2811228               flag_prefix_exa = 1;
2811229             }
2811230           if (isdigit (string[i+1]))
2811231             {
2811232               flag_prefix_oct = 1;
2811233             }
2811234         }
2811235       //
2811236       if (string[i] > '0' && string[i] <= '9')
2811237         {
2811238           flag_prefix_dec = 1;
2811239         }
2811240       //
2811241       // Check compatibility with requested base.
2811242       //
2811243       if (flag_prefix_exa)
2811244         {
2811245           if (base == 0)
2811246             {
2811247               base = 16;
2811248             }
2811249           else if (base == 16)
2811250             {
2811251               ;    // Ok.
2811252             }
2811253           else
2811254             {
2811255               //
2811256               // Incompatible sequence: only the initial zero is reported.
2811257               //
2811258               *endptr = &string[i+1];
2811259               return ((intmax_t) 0);
2811260             }
2811261           //
2811262           // Move on, after the '0x' prefix.
2811263           //
2811264           i += 2;
2811265         }
2811266       //
2811267       if (flag_prefix_oct)
```

```
2811268          {
2811269            if (base == 0)
2811270              {
2811271                base = 8;
2811272              }
2811273            //
2811274            // Move on, after the ‘0’ prefix.
2811275            //
2811276            i += 1;
2811277          }
2811278        //
2811279        if (flag_prefix_dec)
2811280          {
2811281            if (base == 0)
2811282              {
2811283                base = 10;
2811284              }
2811285          }
2811286        //
2811287        // Scan the string.
2811288        //
2811289        for (d = 0, number = 0; d < max_width && string[i] != 0; i++, d++)
2811290          {
2811291            if      (string[i] >= '0' && string[i] <= '9')
2811292              {
2811293                digit = string[i] - '0';
2811294              }
2811295            else if (string[i] >= 'A' && string[i] <= 'F')
2811296              {
2811297                digit = string[i] - 'A' + 10;
2811298              }
2811299            else if (string[i] >= 'a' && string[i] <= 'f')
2811300              {
2811301                digit = string[i] - 'a' + 10;
2811302              }
2811303            else
2811304              {
2811305                digit = 999;
2811306              }
2811307            //
2811308            // Give a sign to the digit.
2811309            //
2811310            digit *= sign;
2811311            //
2811312            // Compare with the base.
2811313            //
2811314            if (base > (digit * sign))
2811315              {
```

```
2811316                 //
2811317                 // Check if the current digit can be safely computed.
2811318                 //
2811319                 previous = number;
2811320                 number  *= base;
2811321                 number  += digit;
2811322                 if (number / base != previous)
2811323                   {
2811324                     //
2811325                     // Out of range.
2811326                     //
2811327                     *endptr = &string[i+1];
2811328                     errset (ERANGE);         // Result too large.
2811329                     if (sign > 0)
2811330                       {
2811331                         return (INTMAX_MAX);
2811332                       }
2811333                     else
2811334                       {
2811335                         return (INTMAX_MIN);
2811336                       }
2811337                   }
2811338               }
2811339             else
2811340               {
2811341                 *endptr = &string[i];
2811342                 return (number);
2811343               }
2811344         }
2811345       //
2811346       // The string is finished or the max digits length is reached.
2811347       //
2811348       *endptr = &string[i];
2811349       //
2811350       return (number);
2811351   }
2811352   //------------------------------------------------------------------------
2811353   static int
2811354   ass_or_eof (int consumed, int assigned)
2811355   {
2811356       if (consumed == 0)
2811357         {
2811358           return (EOF);
2811359         }
2811360       else
2811361         {
2811362           return (assigned);
2811363         }
```

```
2811364 }
2811365 //----------------------------------------------------------------------
```

101.9.39 lib/stdio/vprintf.c

Si veda la sezione 94.128.

```
2820001 #include <stdio.h>
2820002 #include <sys/types.h>
2820003 #include <sys/os16.h>
2820004 #include <string.h>
2820005 #include <unistd.h>
2820006 //----------------------------------------------------------------------
2820007 int
2820008 vprintf (char *restrict format, va_list arg)
2820009 {
2820010     ssize_t     size_written;
2820011     size_t      size;
2820012     size_t      size_total;
2820013     int         status;
2820014     char        string[BUFSIZ];
2820015     char       *buffer = string;
2820016 
2820017     buffer[0] = 0;
2820018     status = vsprintf (buffer, format, arg);
2820019 
2820020     size = strlen (buffer);
2820021     if (size >= BUFSIZ)
2820022       {
2820023         size = BUFSIZ;
2820024       }
2820025 
2820026     for (size_total = 0, size_written = 0;
2820027          size_total < size;
2820028          size_total += size_written, buffer += size_written)
2820029       {
2820030         //
2820031         // Write to the standard output: file descriptor n. 1.
2820032         //
2820033         size_written = write (STDOUT_FILENO, buffer, size - size_total);
2820034         if (size_written < 0)
2820035           {
2820036             return (size_total);
2820037           }
2820038       }
2820039     return (size);
2820040 }
```

101.9.40 lib/stdio/vscanf.c

Si veda la sezione 94.129.

```
2830001 #include <stdio.h>
2830002 //----------------------------------------------------------------------
2830003 int
2830004 vscanf (const char *restrict format, va_list ap)
2830005 {
2830006     return (vfscanf (stdin, format, ap));
2830007 }
2830008 //----------------------------------------------------------------------
```

101.9.41 lib/stdio/vsnprintf.c

Si veda la sezione 94.128.

```
2840001 #include <stdint.h>
2840002 #include <stdbool.h>
2840003 #include <stdlib.h>
2840004 #include <string.h>
2840005 #include <stdio.h>
2840006 //----------------------------------------------------------------------
2840007 static size_t uimaxtoa      (uintmax_t integer, char *buffer, int base,
2840008                              int uppercase, size_t size);
2840009 static size_t imaxtoa       (intmax_t  integer, char *buffer, int base,
2840010                              int uppercase, size_t size);
2840011 static size_t simaxtoa      (intmax_t  integer, char *buffer, int base,
2840012                              int uppercase, size_t size);
2840013 static size_t uimaxtoa_fill (uintmax_t integer, char *buffer, int base,
2840014                              int uppercase, int width, int filler,
2840015                              int max);
2840016 static size_t imaxtoa_fill  (intmax_t  integer, char *buffer, int base,
2840017                              int uppercase, int width, int filler,
2840018                              int max);
2840019 static size_t simaxtoa_fill (intmax_t  integer, char *buffer, int base,
2840020                              int uppercase, int width, int filler,
2840021                              int max);
2840022 static size_t strtostr_fill (char *string, char *buffer, int width,
2840023                              int filler, int max);
2840024 //----------------------------------------------------------------------
2840025 int
2840026 vsnprintf (char *restrict string, size_t size,
2840027            const char *restrict format, va_list ap)
2840028 {
2840029     //
2840030     // We produce at most `size-1' characters, + '\0'.
2840031     // `size' is used also as the max size for internal
2840032     // strings, but only if it is not too big.
```

```
2840033     //
2840034     int         f                     = 0;
2840035     int         s                     = 0;
2840036     int         remain                = size - 1;
2840037     //
2840038     bool        specifier             = 0;
2840039     bool        specifier_flags       = 0;
2840040     bool        specifier_width       = 0;
2840041     bool        specifier_precision   = 0;
2840042     bool        specifier_type        = 0;
2840043     //
2840044     bool        flag_plus             = 0;
2840045     bool        flag_minus            = 0;
2840046     bool        flag_space            = 0;
2840047     bool        flag_alternate        = 0;
2840048     bool        flag_zero             = 0;
2840049     //
2840050     int         alignment;
2840051     int         filler;
2840052     //
2840053     intmax_t    value_i;
2840054     uintmax_t   value_ui;
2840055     char       *value_cp;
2840056     //
2840057     size_t      width;
2840058     size_t      precision;
2840059     #define     str_size  BUFSIZ/2
2840060     char        width_string[str_size];
2840061     char        precision_string[str_size];
2840062     int         w;
2840063     int         p;
2840064     //
2840065     width_string[0]     = '\0';
2840066     precision_string[0] = '\0';
2840067     //
2840068     while (format[f] != 0 && s < (size - 1))
2840069       {
2840070         if (!specifier)
2840071           {
2840072             //------------------- The context is not inside a specifier.
2840073             if (format[f] != '%')
2840074               {
2840075                 string[s] = format[f];
2840076                 s++;
2840077                 remain--;
2840078                 f++;
2840079                 continue;
2840080               }
```

```
2840081                 if (format[f] == '%' && format[f+1] == '%')
2840082                   {
2840083                     string[s] = '%';
2840084                     f++;
2840085                     f++;
2840086                     s++;
2840087                     remain--;
2840088                     continue;
2840089                   }
2840090                 if (format[f] == '%')
2840091                   {
2840092                     f++;
2840093                     specifier = 1;
2840094                     specifier_flags = 1;
2840095                     continue;
2840096                   }
2840097               }
2840098             //
2840099             if (specifier && specifier_flags)
2840100               {
2840101                 //------------------- The context is inside specifier flags.
2840102                 if (format[f] == '+')
2840103                   {
2840104                     flag_plus = 1;
2840105                     f++;
2840106                     continue;
2840107                   }
2840108                 else if (format[f] == '-')
2840109                   {
2840110                     flag_minus = 1;
2840111                     f++;
2840112                     continue;
2840113                   }
2840114                 else if (format[f] == ' ')
2840115                   {
2840116                     flag_space = 1;
2840117                     f++;
2840118                     continue;
2840119                   }
2840120                 else if (format[f] == '#')
2840121                   {
2840122                     flag_alternate = 1;
2840123                     f++;
2840124                     continue;
2840125                   }
2840126                 else if (format[f] == '0')
2840127                   {
2840128                     flag_zero = 1;
```

```
2840129                     f++;
2840130                     continue;
2840131                   }
2840132                 else
2840133                   {
2840134                     specifier_flags = 0;
2840135                     specifier_width = 1;
2840136                   }
2840137               }
2840138             //
2840139             if (specifier && specifier_width)
2840140               {
2840141                 //------------------- The context is inside specifier width.
2840142                 for (w = 0; format[f] >= '0' && format[f] <= '9'
2840143                      && w < str_size; w++)
2840144                   {
2840145                     width_string[w] = format[f];
2840146                     f++;
2840147                   }
2840148                 width_string[w] = '\0';
2840149 
2840150                 specifier_width = 0;
2840151 
2840152                 if (format[f] == '.')
2840153                   {
2840154                     specifier_precision = 1;
2840155                     f++;
2840156                   }
2840157                 else
2840158                   {
2840159                     specifier_precision = 0;
2840160                     specifier_type      = 1;
2840161                   }
2840162               }
2840163             //
2840164             if (specifier && specifier_precision)
2840165               {
2840166                 //--------------- The context is inside specifier precision.
2840167                 for (p = 0; format[f] >= '0' && format[f] <= '9'
2840168                      && p < str_size; p++)
2840169                   {
2840170                     precision_string[p] = format[f];
2840171                     p++;
2840172                   }
2840173                 precision_string[p] = '\0';
2840174 
2840175                 specifier_precision = 0;
2840176                 specifier_type      = 1;
```

```
2840177                 }
2840178             //
2840179             if (specifier && specifier_type)
2840180               {
2840181                 //-------------------- The context is inside specifier type.
2840182                 width     = atoi (width_string);
2840183                 precision = atoi (precision_string);
2840184                                     filler = ' ';
2840185                 if (flag_zero)  filler = '0';
2840186                 if (flag_space) filler = ' ';
2840187                                     alignment = width;
2840188                 if (flag_minus)
2840189                   {
2840190                     alignment = -alignment;
2840191                     filler = ' ';    // The filler character cannot
2840192                                      // be zero, so it is black.
2840193                   }
2840194                 //
2840195                 if (format[f] == 'h' && format[f+1] == 'h')
2840196                   {
2840197                     if (format[f+2] == 'd' || format[f+2] == 'i')
2840198                       {
2840199                         //--------------------------- signed char, base 10.
2840200                         value_i = va_arg (ap, int);
2840201                         if (flag_plus)
2840202                           {
2840203                             s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840204                                                 alignment, filler, remain);
2840205                           }
2840206                         else
2840207                           {
2840208                             s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840209                                                alignment, filler, remain);
2840210                           }
2840211                         f += 3;
2840212                       }
2840213                     else if (format[f+2] == 'u')
2840214                       {
2840215                         //------------------------- unsigned char, base 10.
2840216                         value_ui = va_arg (ap, unsigned int);
2840217                         s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840218                                             alignment, filler, remain);
2840219                         f += 3;
2840220                       }
2840221                     else if (format[f+2] == 'o')
2840222                       {
2840223                         //-------------------------- unsigned char, base 8.
2840224                         value_ui = va_arg (ap, unsigned int);
```

```
2840225                     s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840226                                         alignment, filler, remain);
2840227                     f += 3;
2840228                   }
2840229                 else if (format[f+2] == 'x')
2840230                   {
2840231                     //------------------------- unsigned char, base 16.
2840232                     value_ui = va_arg (ap, unsigned int);
2840233                     s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840234                                         alignment, filler, remain);
2840235                     f += 3;
2840236                   }
2840237                 else if (format[f+2] == 'X')
2840238                   {
2840239                     //------------------------- unsigned char, base 16.
2840240                     value_ui = va_arg (ap, unsigned int);
2840241                     s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840242                                         alignment, filler, remain);
2840243                     f += 3;
2840244                   }
2840245                 else
2840246                   {
2840247                     //---------------- unsupported or unknown specifier.
2840248                     f += 2;
2840249                   }
2840250               }
2840251             else if (format[f] == 'h')
2840252               {
2840253                 if (format[f+1] == 'd' || format[f+1] == 'i')
2840254                   {
2840255                     //----------------------------- short int, base 10.
2840256                     value_i = va_arg (ap, int);
2840257                     if (flag_plus)
2840258                       {
2840259                         s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840260                                             alignment, filler, remain);
2840261                       }
2840262                     else
2840263                       {
2840264                         s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840265                                            alignment, filler, remain);
2840266                       }
2840267                     f += 2;
2840268                   }
2840269                 else if (format[f+1] == 'u')
2840270                   {
2840271                     //-------------------- unsigned short int, base 10.
2840272                     value_ui = va_arg (ap, unsigned int);
```

```
2840273                         s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840274                                             alignment, filler, remain);
2840275                         f += 2;
2840276                       }
2840277                     else if (format[f+1] == 'o')
2840278                       {
2840279                         //--------------------- unsigned short int, base 8.
2840280                         value_ui = va_arg (ap, unsigned int);
2840281                         s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840282                                             alignment, filler, remain);
2840283                         f += 2;
2840284                       }
2840285                     else if (format[f+1] == 'x')
2840286                       {
2840287                         //-------------------- unsigned short int, base 16.
2840288                         value_ui = va_arg (ap, unsigned int);
2840289                         s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840290                                             alignment, filler, remain);
2840291                         f += 2;
2840292                       }
2840293                     else if (format[f+1] == 'X')
2840294                       {
2840295                         //-------------------- unsigned short int, base 16.
2840296                         value_ui = va_arg (ap, unsigned int);
2840297                         s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840298                                             alignment, filler, remain);
2840299                         f += 2;
2840300                       }
2840301                     else
2840302                       {
2840303                         //---------------- unsupported or unknown specifier.
2840304                         f += 1;
2840305                       }
2840306                   }
2840307 
2840308                     //-------------------------------------------------------
2840309                     // There is no `long long int'.
2840310                     //-------------------------------------------------------
2840311 
2840312                 else if (format[f] == 'l')
2840313                   {
2840314                     if (format[f+1] == 'd' || format[f+1] == 'i')
2840315                       {
2840316                         //--------------------------------- long int base 10.
2840317                         value_i = va_arg (ap, long int);
2840318                         if (flag_plus)
2840319                           {
2840320                             s += simaxtoa_fill (value_i, &string[s], 10, 0,
```

```
2840321                                              alignment, filler, remain);
2840322                       }
2840323                     else
2840324                       {
2840325                         s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840326                                             alignment, filler, remain);
2840327                       }
2840328                     f += 2;
2840329                   }
2840330                 else if (format[f+1] == 'u')
2840331                   {
2840332                     //---------------------- Unsigned long int base 10.
2840333                     value_ui = va_arg (ap, unsigned long int);
2840334                     s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840335                                         alignment, filler, remain);
2840336                     f += 2;
2840337                   }
2840338                 else if (format[f+1] == 'o')
2840339                   {
2840340                     //----------------------- Unsigned long int base 8.
2840341                     value_ui = va_arg (ap, unsigned long int);
2840342                     s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840343                                         alignment, filler, remain);
2840344                     f += 2;
2840345                   }
2840346                 else if (format[f+1] == 'x')
2840347                   {
2840348                     //---------------------- Unsigned long int base 16.
2840349                     value_ui = va_arg (ap, unsigned long int);
2840350                     s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840351                                         alignment, filler, remain);
2840352                     f += 2;
2840353                   }
2840354                 else if (format[f+1] == 'X')
2840355                   {
2840356                     //---------------------- Unsigned long int base 16.
2840357                     value_ui = va_arg (ap, unsigned long int);
2840358                     s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840359                                         alignment, filler, remain);
2840360                     f += 2;
2840361                   }
2840362                 else
2840363                   {
2840364                     //---------------- unsupported or unknown specifier.
2840365                     f += 1;
2840366                   }
2840367               }
2840368             else if (format[f] == 'j')
```

```
2840369                 {
2840370                   if (format[f+1] == 'd' || format[f+1] == 'i')
2840371                     {
2840372                       //------------------------------- intmax_t base 10.
2840373                       value_i = va_arg (ap, intmax_t);
2840374                       if (flag_plus)
2840375                         {
2840376                           s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840377                                               alignment, filler, remain);
2840378                         }
2840379                       else
2840380                         {
2840381                           s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840382                                              alignment, filler, remain);
2840383                         }
2840384                       f += 2;
2840385                     }
2840386                   else if (format[f+1] == 'u')
2840387                     {
2840388                       //------------------------------ uintmax_t base 10.
2840389                       value_ui = va_arg (ap, uintmax_t);
2840390                       s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840391                                           alignment, filler, remain);
2840392                       f += 2;
2840393                     }
2840394                   else if (format[f+1] == 'o')
2840395                     {
2840396                       //------------------------------- uintmax_t base 8.
2840397                       value_ui = va_arg (ap, uintmax_t);
2840398                       s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840399                                           alignment, filler, remain);
2840400                       f += 2;
2840401                     }
2840402                   else if (format[f+1] == 'x')
2840403                     {
2840404                       //------------------------------ uintmax_t base 16.
2840405                       value_ui = va_arg (ap, uintmax_t);
2840406                       s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840407                                           alignment, filler, remain);
2840408                       f += 2;
2840409                     }
2840410                   else if (format[f+1] == 'X')
2840411                     {
2840412                       //------------------------------ uintmax_t base 16.
2840413                       value_ui = va_arg (ap, uintmax_t);
2840414                       s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840415                                           alignment, filler, remain);
2840416                       f += 2;
```

```
2840417                   }
2840418                 else
2840419                   {
2840420                     //---------------- unsupported or unknown specifier.
2840421                     f += 1;
2840422                   }
2840423               }
2840424             else if (format[f] == 'z')
2840425               {
2840426                 if (format[f+1] == 'd'
2840427                     || format[f+1] == 'i'
2840428                     || format[f+1] == 'i')
2840429                   {
2840430                     //---------------------------------- size_t base 10.
2840431                     value_ui = va_arg (ap, unsigned long int);
2840432                     s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840433                                         alignment, filler, remain);
2840434                     f += 2;
2840435                   }
2840436                 else if (format[f+1] == 'o')
2840437                   {
2840438                     //----------------------------------- size_t base 8.
2840439                     value_ui = va_arg (ap, unsigned long int);
2840440                     s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840441                                         alignment, filler, remain);
2840442                     f += 2;
2840443                   }
2840444                 else if (format[f+1] == 'x')
2840445                   {
2840446                     //---------------------------------- size_t base 16.
2840447                     value_ui = va_arg (ap, unsigned long int);
2840448                     s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840449                                         alignment, filler, remain);
2840450                     f += 2;
2840451                   }
2840452                 else if (format[f+1] == 'X')
2840453                   {
2840454                     //---------------------------------- size_t base 16.
2840455                     value_ui = va_arg (ap, unsigned long int);
2840456                     s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840457                                         alignment, filler, remain);
2840458                     f += 2;
2840459                   }
2840460                 else
2840461                   {
2840462                     //---------------- unsupported or unknown specifier.
2840463                     f += 1;
2840464                   }
```

```
2840465                    }
2840466                  else if (format[f] == 't')
2840467                    {
2840468                      if (format[f+1] == 'd' || format[f+1] == 'i')
2840469                        {
2840470                          //------------------------------ ptrdiff_t base 10.
2840471                          value_i = va_arg (ap, long int);
2840472                          if (flag_plus)
2840473                            {
2840474                              s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840475                                                  alignment, filler, remain);
2840476                            }
2840477                          else
2840478                            {
2840479                              s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840480                                                 alignment, filler, remain);
2840481                            }
2840482                          f += 2;
2840483                        }
2840484                      else if (format[f+1] == 'u')
2840485                        {
2840486                          //---------------- ptrdiff_t base 10, without sign.
2840487                          value_ui = va_arg (ap, unsigned long int);
2840488                          s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840489                                              alignment, filler, remain);
2840490                          f += 2;
2840491                        }
2840492                      else if (format[f+1] == 'o')
2840493                        {
2840494                          //----------------- ptrdiff_t base 8, without sign.
2840495                          value_ui = va_arg (ap, unsigned long int);
2840496                          s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840497                                              alignment, filler, remain);
2840498                          f += 2;
2840499                        }
2840500                      else if (format[f+1] == 'x')
2840501                        {
2840502                          //---------------- ptrdiff_t base 16, without sign.
2840503                          value_ui = va_arg (ap, unsigned long int);
2840504                          s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840505                                              alignment, filler, remain);
2840506                          f += 2;
2840507                        }
2840508                      else if (format[f+1] == 'X')
2840509                        {
2840510                          //---------------- ptrdiff_t base 16, without sign.
2840511                          value_ui = va_arg (ap, unsigned long int);
2840512                          s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
```

```
2840513                                         alignment, filler, remain);
2840514                       f += 2;
2840515                     }
2840516                   else
2840517                     {
2840518                       //---------------- unsupported or unknown specifier.
2840519                       f += 1;
2840520                     }
2840521                 }
2840522               if (format[f] == 'd' || format[f] == 'i')
2840523                 {
2840524                   //---------------------------------------- int base 10.
2840525                   value_i = va_arg (ap, int);
2840526                   if (flag_plus)
2840527                     {
2840528                       s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840529                                         alignment, filler, remain);
2840530                     }
2840531                   else
2840532                     {
2840533                       s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840534                                        alignment, filler, remain);
2840535                     }
2840536                   f += 1;
2840537                 }
2840538               else if (format[f] == 'u')
2840539                 {
2840540                   //------------------------------- unsigned int base 10.
2840541                   value_ui = va_arg (ap, unsigned int);
2840542                   s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840543                                     alignment, filler, remain);
2840544                   f += 1;
2840545                 }
2840546               else if (format[f] == 'o')
2840547                 {
2840548                   //-------------------------------- unsigned int base 8.
2840549                   value_ui = va_arg (ap, unsigned int);
2840550                   s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840551                                     alignment, filler, remain);
2840552                   f += 1;
2840553                 }
2840554               else if (format[f] == 'x')
2840555                 {
2840556                   //------------------------------- unsigned int base 16.
2840557                   value_ui = va_arg (ap, unsigned int);
2840558                   s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840559                                     alignment, filler, remain);
2840560                   f += 1;
```

```
2840561                   }
2840562                 else if (format[f] == 'X')
2840563                   {
2840564                     //------------------------------ unsigned int base 16.
2840565                     value_ui = va_arg (ap, unsigned int);
2840566                     s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840567                                         alignment, filler, remain);
2840568                     f += 1;
2840569                   }
2840570                 else if (format[f] == 'c')
2840571                   {
2840572                     //------------------------------------- unsigned char.
2840573                     value_ui = va_arg (ap, unsigned int);
2840574                     string[s] = (char) value_ui;
2840575                     s += 1;
2840576                     f += 1;
2840577                   }
2840578                 else if (format[f] == 's')
2840579                   {
2840580                     //-------------------------------------------- string.
2840581                     value_cp = va_arg (ap, char *);
2840582                     filler = ' ';
2840583
2840584                     s += strtostr_fill (value_cp, &string[s], alignment,
2840585                                         filler, remain);
2840586                     f += 1;
2840587                   }
2840588                 else
2840589                   {
2840590                     //------------------- unsupported or unknown specifier.
2840591                     ;
2840592                   }
2840593                 //------------------------------------------------------------
2840594                 // End of specifier.
2840595                 //------------------------------------------------------------
2840596                 width_string[0]     = '\0';
2840597                 precision_string[0] = '\0';
2840598
2840599                 specifier           = 0;
2840600                 specifier_flags     = 0;
2840601                 specifier_width     = 0;
2840602                 specifier_precision = 0;
2840603                 specifier_type      = 0;
2840604
2840605                 flag_plus           = 0;
2840606                 flag_minus          = 0;
2840607                 flag_space          = 0;
2840608                 flag_alternate      = 0;
```

```
2840609               flag_zero           = 0;
2840610             }
2840611         }
2840612     string[s] = '\0';
2840613     return s;
2840614 }
2840615 //----------------------------------------------------------------------
2840616 // Static functions.
2840617 //----------------------------------------------------------------------
2840618 static size_t
2840619 uimaxtoa (uintmax_t integer, char *buffer, int base, int uppercase,
2840620           size_t size)
2840621 {
2840622     //------------------------------------------------------------------
2840623     // Convert a maximum rank integer into a string.
2840624     //------------------------------------------------------------------
2840625 
2840626         uintmax_t  integer_copy = integer;
2840627            size_t  digits;
2840628               int  b;
2840629     unsigned char  remainder;
2840630 
2840631     for (digits = 0; integer_copy > 0; digits++)
2840632       {
2840633         integer_copy = integer_copy / base;
2840634       }
2840635 
2840636     if (buffer == NULL && integer == 0) return 1;
2840637     if (buffer == NULL && integer > 0)  return digits;
2840638 
2840639     if (integer == 0)
2840640       {
2840641         buffer[0] = '0';
2840642         buffer[1] = '\0';
2840643         return 1;
2840644       }
2840645     //
2840646     // Fix the maximum number of digits.
2840647     //
2840648     if (size > 0 && digits > size) digits = size;
2840649     //
2840650     *(buffer + digits) = '\0';            // End of string.
2840651 
2840652     for (b = digits - 1; integer != 0 && b >= 0; b--)
2840653       {
2840654         remainder = integer % base;
2840655         integer   = integer / base;
2840656 
```

```
2840657          if (remainder <= 9)
2840658            {
2840659              *(buffer + b) = remainder + '0';
2840660            }
2840661          else
2840662            {
2840663              if (uppercase)
2840664                {
2840665                  *(buffer + b) = remainder - 10 + 'A';
2840666                }
2840667              else
2840668                {
2840669                  *(buffer + b) = remainder - 10 + 'a';
2840670                }
2840671            }
2840672        }
2840673      return digits;
2840674  }
2840675  //----------------------------------------------------------------------
2840676  static size_t
2840677  imaxtoa (intmax_t integer, char *buffer, int base, int uppercase,
2840678           size_t size)
2840679  {
2840680      //------------------------------------------------------------------
2840681      // Convert a maximum rank integer with sign into a string.
2840682      //------------------------------------------------------------------
2840683
2840684      if (integer >= 0)
2840685        {
2840686          return uimaxtoa (integer, buffer, base, uppercase, size);
2840687        }
2840688      //
2840689      // At this point, there is a negative number, less than zero.
2840690      //
2840691      if (buffer == NULL)
2840692        {
2840693          return uimaxtoa (-integer, NULL, base, uppercase, size) + 1;
2840694        }
2840695
2840696      *buffer = '-';          // The minus sign is needed at the beginning.
2840697      if (size == 1)
2840698        {
2840699          *(buffer + 1) = '\0';
2840700          return 1;
2840701        }
2840702      else
2840703        {
2840704          return uimaxtoa (-integer, buffer+1, base, uppercase, size-1)
```

```
2840705                  + 1;
2840706        }
2840707 }
2840708 //----------------------------------------------------------------------
2840709 static size_t
2840710 simaxtoa (intmax_t integer, char *buffer, int base, int uppercase,
2840711           size_t size)
2840712 {
2840713     //------------------------------------------------------------------
2840714     // Convert a maximum rank integer with sign into a string, placing
2840715     // the sign also if it is positive.
2840716     //------------------------------------------------------------------
2840717 
2840718     if (buffer == NULL && integer >= 0)
2840719       {
2840720         return uimaxtoa (integer, NULL, base, uppercase, size) + 1;
2840721       }
2840722 
2840723     if (buffer == NULL && integer < 0)
2840724       {
2840725         return uimaxtoa (-integer, NULL, base, uppercase, size) + 1;
2840726       }
2840727     //
2840728     // At this point, 'buffer' is different from NULL.
2840729     //
2840730     if (integer >= 0)
2840731       {
2840732         *buffer = '+';
2840733       }
2840734     else
2840735       {
2840736         *buffer = '-';
2840737       }
2840738 
2840739     if (size == 1)
2840740       {
2840741         *(buffer + 1) = '\0';
2840742         return 1;
2840743       }
2840744 
2840745     if (integer >= 0)
2840746       {
2840747         return uimaxtoa (integer, buffer+1, base, uppercase, size-1)
2840748                + 1;
2840749       }
2840750     else
2840751       {
2840752         return uimaxtoa (-integer, buffer+1, base, uppercase, size-1)
```

```
2840753                 + 1;
2840754       }
2840755 }
2840756 //----------------------------------------------------------------------
2840757 static size_t
2840758 uimaxtoa_fill (uintmax_t integer, char *buffer, int base,
2840759                int uppercase, int width, int filler, int max)
2840760 {
2840761     //------------------------------------------------------------------
2840762     // Convert a maximum rank integer without sign into a string,
2840763     // takeing care of the alignment.
2840764     //------------------------------------------------------------------
2840765 
2840766     size_t size_i;
2840767     size_t size_f;
2840768 
2840769     if (max < 0) return 0; // «max» deve essere un valore positivo.
2840770 
2840771     size_i = uimaxtoa (integer, NULL, base, uppercase, 0);
2840772 
2840773     if (width > 0 &&  max > 0 && width >  max) width =  max;
2840774     if (width < 0 && -max < 0 && width < -max) width = -max;
2840775 
2840776     if (size_i > abs (width))
2840777       {
2840778         return uimaxtoa (integer, buffer, base, uppercase, abs (width));
2840779       }
2840780 
2840781     if (width == 0 && max > 0)
2840782       {
2840783         return uimaxtoa (integer, buffer, base, uppercase, max);
2840784       }
2840785 
2840786     if (width == 0)
2840787       {
2840788         return uimaxtoa (integer, buffer, base, uppercase, abs (width));
2840789       }
2840790     //
2840791     // size_i <= abs (width).
2840792     //
2840793     size_f = abs (width) - size_i;
2840794 
2840795     if (width < 0)
2840796       {
2840797         // Left alignment.
2840798         uimaxtoa (integer, buffer, base, uppercase, 0);
2840799         memset (buffer + size_i, filler, size_f);
2840800       }
```

```
2840801     else
2840802       {
2840803         // Right alignment.
2840804         memset (buffer, filler, size_f);
2840805         uimaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840806       }
2840807     *(buffer + abs (width)) = '\0';
2840808 
2840809     return abs (width);
2840810 }
2840811 //----------------------------------------------------------------------
2840812 static size_t
2840813 imaxtoa_fill (intmax_t integer, char *buffer, int base,
2840814               int uppercase, int width, int filler, int max)
2840815 {
2840816     //------------------------------------------------------------------
2840817     // Convert a maximum rank integer with sign into a string,
2840818     // takeing care of the alignment.
2840819     //------------------------------------------------------------------
2840820 
2840821     size_t size_i;
2840822     size_t size_f;
2840823 
2840824     if (max < 0) return 0; // 'max' must be a positive value.
2840825 
2840826     size_i = imaxtoa (integer, NULL, base, uppercase, 0);
2840827 
2840828     if (width > 0 &&  max > 0 && width >  max) width =  max;
2840829     if (width < 0 && -max < 0 && width < -max) width = -max;
2840830 
2840831     if (size_i > abs (width))
2840832       {
2840833         return imaxtoa (integer, buffer, base, uppercase, abs (width));
2840834       }
2840835 
2840836     if (width == 0 && max > 0)
2840837       {
2840838         return imaxtoa (integer, buffer, base, uppercase, max);
2840839       }
2840840 
2840841     if (width == 0)
2840842       {
2840843         return imaxtoa (integer, buffer, base, uppercase, abs (width));
2840844       }
2840845 
2840846     // size_i <= abs (width).
2840847 
2840848     size_f = abs (width) - size_i;
```

```
2840849
2840850     if (width < 0)
2840851       {
2840852         // Left alignment.
2840853         imaxtoa (integer, buffer, base, uppercase, 0);
2840854         memset (buffer + size_i, filler, size_f);
2840855       }
2840856     else
2840857       {
2840858         // Right alignment.
2840859         memset (buffer, filler, size_f);
2840860         imaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840861       }
2840862     *(buffer + abs (width)) = '\0';
2840863
2840864     return abs (width);
2840865 }
2840866 //----------------------------------------------------------------------
2840867 static size_t
2840868 simaxtoa_fill (intmax_t integer, char *buffer, int base,
2840869                int uppercase, int width, int filler, int max)
2840870 {
2840871     //------------------------------------------------------------------
2840872     // Convert a maximum rank integer with sign into a string,
2840873     // placing the sign also if it is positive and takeing care of the
2840874     // alignment.
2840875     //------------------------------------------------------------------
2840876
2840877     size_t size_i;
2840878     size_t size_f;
2840879
2840880     if (max < 0) return 0; // 'max' must be a positive value.
2840881
2840882     size_i = simaxtoa (integer, NULL, base, uppercase, 0);
2840883
2840884     if (width > 0 &&  max > 0 && width >  max) width =  max;
2840885     if (width < 0 && -max < 0 && width < -max) width = -max;
2840886
2840887     if (size_i > abs (width))
2840888       {
2840889         return simaxtoa (integer, buffer, base, uppercase, abs (width));
2840890       }
2840891
2840892     if (width == 0 && max > 0)
2840893       {
2840894         return simaxtoa (integer, buffer, base, uppercase, max);
2840895       }
2840896
```

```
2840897     if (width == 0)
2840898       {
2840899         return simaxtoa (integer, buffer, base, uppercase, abs (width));
2840900       }
2840901     //
2840902     // size_i <= abs (width).
2840903     //
2840904     size_f = abs (width) - size_i;
2840905 
2840906     if (width < 0)
2840907       {
2840908         // Left alignment.
2840909         simaxtoa (integer, buffer, base, uppercase, 0);
2840910         memset (buffer + size_i, filler, size_f);
2840911       }
2840912     else
2840913       {
2840914         // Right alignment.
2840915         memset (buffer, filler, size_f);
2840916         simaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840917       }
2840918     *(buffer + abs (width)) = '\0';
2840919 
2840920     return abs (width);
2840921 }
2840922 //----------------------------------------------------------------------
2840923 static size_t
2840924 strtostr_fill (char *string, char *buffer, int width, int filler,
2840925                int max)
2840926 {
2840927     //------------------------------------------------------------------
2840928     // Transfer a string with care for the alignment.
2840929     //------------------------------------------------------------------
2840930 
2840931     size_t size_s;
2840932     size_t size_f;
2840933 
2840934     if (max < 0) return 0; // 'max' must be a positive value.
2840935 
2840936     size_s = strlen (string);
2840937 
2840938     if (width > 0 &&  max > 0 && width >  max) width =  max;
2840939     if (width < 0 && -max < 0 && width < -max) width = -max;
2840940 
2840941     if (width != 0 && size_s > abs (width))
2840942       {
2840943         memcpy (buffer, string, abs (width));
2840944         buffer[width] = '\0';
```

```
2840945             return width;
2840946           }
2840947 
2840948         if (width == 0 && max > 0 && size_s > max)
2840949           {
2840950             memcpy (buffer, string, max);
2840951             buffer[max] = '\0';
2840952             return max;
2840953           }
2840954 
2840955         if (width == 0 && max > 0 && size_s < max)
2840956           {
2840957             memcpy (buffer, string, size_s);
2840958             buffer[size_s] = '\0';
2840959             return size_s;
2840960           }
2840961         //
2840962         // width =! 0
2840963         // size_s <= abs (width)
2840964         //
2840965         size_f = abs (width) - size_s;
2840966 
2840967         if (width < 0)
2840968           {
2840969             // Right alignment.
2840970             memset (buffer, filler, size_f);
2840971             strncpy (buffer+size_f, string, size_s);
2840972           }
2840973         else
2840974           {
2840975             // Left alignment.
2840976             strncpy (buffer, string, size_s);
2840977             memset (buffer+size_s, filler, size_f);
2840978           }
2840979         *(buffer + abs (width)) = '\0';
2840980 
2840981         return abs (width);
2840982     }
2840983 
2840984 
```

101.9.42 lib/stdio/vsprintf.c

«

Si veda la sezione 94.128.

```
2850001 #include <stdio.h>
2850002 #include <sys/os16.h>
2850003 //----------------------------------------------------------------------
2850004 int
2850005 vsprintf (char *string, char *restrict format, va_list arg)
2850006 {
2850007     return (vsnprintf (string, BUFSIZ, format, arg));
2850008 }
```

101.9.43 lib/stdio/vsscanf.c

«

Si veda la sezione 94.129.

```
2860001 #include <stdio.h>
2860002
2860003 //----------------------------------------------------------------------
2860004 int vfsscanf (FILE *restrict fp, const char *string,
2860005               const char *restrict format, va_list ap);
2860006 //----------------------------------------------------------------------
2860007 int
2860008 vsscanf (const char *string, const char *restrict format, va_list ap)
2860009 {
2860010     return (vfsscanf (NULL, string, format, ap));
2860011 }
2860012 //----------------------------------------------------------------------
```

101.10 os16: «lib/stdlib.h»

«

Si veda la sezione 97.2.

```
2870001 #ifndef _STDLIB_H
2870002 #define _STDLIB_H       1
2870003 //----------------------------------------------------------------------
2870004
2870005 #include <size_t.h>
2870006 #include <wchar_t.h>
2870007 #include <NULL.h>
2870008 #include <limits.h>
2870009 #include <const.h>
2870010 #include <restrict.h>
2870011 //----------------------------------------------------------------------
2870012 typedef struct {
2870013     int      quot;
2870014     int      rem;
```

```
2870015 } div_t;
2870016 //----------------------------------------------------------------------
2870017 typedef struct {
2870018     long int quot;
2870019     long int rem;
2870020 } ldiv_t;
2870021 //----------------------------------------------------------------------
2870022 typedef void (*atexit_t) (void);        // Non standard. [1]
2870023 //
2870024 // [1] The type 'atexit_t' is a pointer to a function for the "at exit"
2870025 //     procedure, with no parameters and returning void. With the
2870026 //     declaration of type 'atexit_t', the function prototype of
2870027 //     'atexit()' is easier to declare and to understand. Original
2870028 //     declaration is:
2870029 //
2870030 //              int atexit (void (*function) (void));
2870031 //
2870032 //----------------------------------------------------------------------
2870033 #define EXIT_FAILURE    1
2870034 #define EXIT_SUCCESS    0
2870035 #define RAND_MAX        INT_MAX
2870036 #define MB_CUR_MAX      ((size_t) MB_LEN_MAX)
2870037 //----------------------------------------------------------------------
2870038 void              _Exit    (int status);
2870039 void              abort    (void);
2870040 int               abs      (int j);
2870041 int               atexit   (atexit_t function);
2870042 int               atoi     (const char *string);
2870043 long int          atol     (const char *string);
2870044 //void           *bsearch  (const void *key, const void *base,
2870045 //                          size_t nmemb, size_t size,
2870046 //                          int (*compar) (const void *, const void *));
2870047 #define           calloc(b, s) (malloc ((b) * (s)))
2870048 div_t             div      (int numer, int denom);
2870049 void              exit     (int status);
2870050 void              free     (void *ptr);
2870051 char             *getenv   (const char *name);
2870052 long int          labs     (long int j);
2870053 ldiv_t            ldiv     (long int numer, long int denom);
2870054 void             *malloc   (size_t size);
2870055 int               putenv   (const char *string);
2870056 void              qsort    (void *base, size_t nmemb, size_t size,
2870057                             int (*compare) (const void *,
2870058                                             const void *));
2870059 int               rand     (void);
2870060 void             *realloc  (void *ptr, size_t size);
2870061 int               setenv   (const char *name, const char *value,
2870062                             int overwrite);
```

```
2870063 void                srand    (unsigned int seed);
2870064 long int            strtol   (const char *restrict string,
2870065                               char **restrict endptr, int base);
2870066 unsigned long int strtoul  (const char * restrict string,
2870067                               char ** restrict endptr, int base);
2870068 //int               system   (const char *string);
2870069 int                 unsetenv (const char *name);
2870070 
2870071 #endif
```

101.10.1 lib/stdlib/_Exit.c

«

Si veda la sezione 93.2.

```
2880001 #include <stdlib.h>
2880002 #include <sys/os16.h>
2880003 //----------------------------------------------------------------------
2880004 void
2880005 _Exit (int status)
2880006 {
2880007     sysmsg_exit_t msg;
2880008     //
2880009     // Only the low eight bit are returned.
2880010     //
2880011     msg.status = (status & 0xFF);
2880012     //
2880013     //
2880014     //
2880015     sys (SYS_EXIT, &msg, (sizeof msg));
2880016     //
2880017     // Should not return from system call, but if it does, loop
2880018     // forever:
2880019     //
2880020     while (1);
2880021 }
```

101.10.2 lib/stdlib/abort.c

«

Si veda la sezione 94.2.

```
2890001 #include <stdlib.h>
2890002 #include <sys/types.h>
2890003 #include <signal.h>
2890004 #include <unistd.h>
2890005 //----------------------------------------------------------------------
2890006 void
2890007 abort (void)
```

```
2890008 {
2890009     pid_t        pid;
2890010     sighandler_t sig_previous;
2890011     //
2890012     // Set `SIGABRT' to a default action.
2890013     //
2890014     sig_previous = signal (SIGABRT, SIG_DFL);
2890015     //
2890016     // If the previous action was something different than symbolic
2890017     // ones, configure again the previous action.
2890018     //
2890019     if (sig_previous != SIG_DFL &&
2890020         sig_previous != SIG_IGN &&
2890021         sig_previous != SIG_ERR)
2890022       {
2890023         signal (SIGABRT, sig_previous);
2890024       }
2890025     //
2890026     // Get current process ID and sent the signal.
2890027     //
2890028     pid = getpid ();
2890029     kill (pid, SIGABRT);
2890030     //
2890031     // Second chance
2890032     //
2890033     for (;;)
2890034       {
2890035         signal (SIGABRT, SIG_DFL);
2890036         pid = getpid ();
2890037         kill (pid, SIGABRT);
2890038       }
2890039 }
```

101.10.3 lib/stdlib/abs.c

«

Si veda la sezione 94.3.

```
2900001 #include <stdlib.h>
2900002 //----------------------------------------------------------------------
2900003 int
2900004 abs (int j)
2900005 {
2900006     if (j < 0)
2900007       {
2900008         return -j;
2900009       }
2900010     else
2900011       {
```

```
2900012             return j;
2900013           }
2900014   }
```

101.10.4 lib/stdlib/alloc.c

«

Si veda la sezione 94.66.

```
2910001 #include <stdlib.h>
2910002 #include <string.h>
2910003 #include <errno.h>
2910004 #include <limits.h>
2910005 #include <stdio.h>
2910006 //----------------------------------------------------------------------
2910007 #define MEMORY_BLOCK_SIZE       1024
2910008 //----------------------------------------------------------------------
2910009 static char     _alloc_memory[LONG_BIT][MEMORY_BLOCK_SIZE];   // [1]
2910010 static size_t   _alloc_size[LONG_BIT];                        // [2]
2910011 static long int _alloc_map;                                   // [3]
2910012 //
2910013 // [1] Memory to be allocated.
2910014 // [2] Sizes allocated.
2910015 // [3] Memory block map. The memory map is made of a single integer and
2910016 //     the rightmost bit is the first memory block.
2910017 //----------------------------------------------------------------------
2910018 void *
2910019 malloc (size_t size)
2910020 {
2910021     size_t   size_free;    // Size free found that might be allocated.
2910022     int      m;            // Index inside '_alloc_memory[][]' table.
2910023     int      s;            // Start index for a free memory area.
2910024     long int mask;         // Mask to compare with '_alloc_map'.
2910025     long int alloc;        // New allocation map.
2910026     //
2910027     // Check for arguments.
2910028     //
2910029     if (size == 0)
2910030       {
2910031         return (NULL);
2910032       }
2910033     //
2910034     for (s = 0, m = 0; m < LONG_BIT; m++)
2910035       {
2910036         mask = 1;
2910037         mask <<= m;
2910038         //
2910039         if (_alloc_map & mask)
2910040           {
```

```
2910041                 //
2910042                 // The memory block is not free.
2910043                 //
2910044                 s          = m + 1;
2910045                 size_free = 0;
2910046                 alloc     = 0;
2910047               }
2910048             else
2910049               {
2910050                 alloc     |= mask;
2910051                 size_free += MEMORY_BLOCK_SIZE;
2910052               }
2910053             if (size_free >= size)
2910054               {
2910055                 //
2910056                 // Space found: update '_alloc_size[]' table, the map inside
2910057                 // '_alloc_map' and return the memory address.
2910058                 //
2910059                 _alloc_size[s] = size_free;
2910060                 _alloc_map    |= alloc;
2910061                 return ((void *) &_alloc_memory[s][0]);
2910062               }
2910063         }
2910064     //
2910065     // No space left.
2910066     //
2910067     errset (ENOMEM);                    // Not enough space.
2910068     //
2910069     return (NULL);
2910070 }
2910071 //----------------------------------------------------------------------
2910072 void
2910073 free (void *address)
2910074 {
2910075     size_t   size_free;  // Size to make free.
2910076     int      m;          // Index inside '_alloc_memory[][]' table.
2910077     int      s;          // Start index.
2910078     long int mask;       // Mask to compare with '_alloc_map'.
2910079     long int alloc;      // New allocation map.
2910080     //
2910081     // Check argument.
2910082     //
2910083     if (address == NULL)
2910084       {
2910085         return;
2910086       }
2910087     //
2910088     // Find the original allocated address inside '_alloc_memory[][]'
```

```
2910089      // table.
2910090      //
2910091      for (m = 0; m < LONG_BIT; m++)
2910092        {
2910093          if (address == (void *) &_alloc_memory[m][0])
2910094            {
2910095              //
2910096              // This is the right memory block.
2910097              //
2910098              if (_alloc_size[m] == 0)
2910099                {
2910100                  //
2910101                  // The block found is not allocated.
2910102                  //
2910103                  return;
2910104                }
2910105              else
2910106                {
2910107                  //
2910108                  // Build the map of the memory to set free.
2910109                  //
2910110                  size_free = _alloc_size[m];
2910111                  for (alloc = 0, s = m;
2910112                       size_free > 0 && s < LONG_BIT;
2910113                       size_free -= MEMORY_BLOCK_SIZE, s++)
2910114                    {
2910115                      mask   = 1;
2910116                      mask <<= s;
2910117                      alloc |= mask;
2910118                    }
2910119                  //
2910120                  // Compare the map of memory to be freed with the
2910121                  // reality allocated one, then free the memory.
2910122                  //
2910123                  if ((_alloc_map & alloc) == alloc)
2910124                    {
2910125                      _alloc_map    &= ~alloc;
2910126                      _alloc_size[m] = 0;
2910127                      return;
2910128                    }
2910129                  //
2910130                  // The real map does not report the same amount of
2910131                  // allocated memory, so nothing is freed.
2910132                  //
2910133                  return;
2910134                }
2910135            }
2910136        }
```

```
2910137     //
2910138     // Address not allocated.
2910139     //
2910140     return;
2910141 }
2910142 //----------------------------------------------------------------------
2910143 void *
2910144 realloc (void *address, size_t size)
2910145 {
2910146     char   *address_new;
2910147     char   *address_old = (char *) address;
2910148     size_t size_old    = 0;
2910149     size_t size_new    = size;
2910150     int    m;                   // Index inside the memory table;
2910151     //
2910152     // Check arguments.
2910153     //
2910154     if (size == 0)              return (NULL);
2910155     if (address == NULL)        return (malloc (size));
2910156     //
2910157     // Locate original allocation.
2910158     //
2910159     for (m = 0; m < LONG_BIT; m++)
2910160       {
2910161         if (address_old == (char *) &_alloc_memory[m][0])
2910162           {
2910163             size_old = _alloc_size[m];
2910164             break;
2910165           }
2910166       }
2910167     //
2910168     // Check if a valid size was found.
2910169     //
2910170     if (size_old == 0)
2910171       {
2910172         //
2910173         // Address not found or size not valid.
2910174         //
2910175         return (NULL);
2910176       }
2910177     //
2910178     // Allocate the new memory.
2910179     //
2910180     address_new = malloc (size);
2910181     //
2910182     // Check allocation. If there is an error, the variable ‘errno’
2910183     // is already updated by ‘malloc()’.
2910184     //
```

```
2910185     if (address_new == NULL)
2910186       {
2910187         return (NULL);
2910188       }
2910189     //
2910190     // Copy old memory.
2910191     //
2910192     for (; size_old > 0 && size_new > 0;
2910193          size_old--, size_new--, address_new++, address_old++)
2910194       {
2910195         *address_new = *address_old;
2910196       }
2910197     //
2910198     // Free old memory.
2910199     //
2910200     free (address);
2910201     //
2910202     // Return the new address.
2910203     //
2910204     return (address_new);
2910205 }
2910206 //----------------------------------------------------------------------
```

101.10.5 lib/stdlib/atexit.c

«

Si veda la sezione 94.4.

```
2920001 #include <stdlib.h>
2920002 #include <stdio.h>
2920003 //----------------------------------------------------------------------
2920004 atexit_t _atexit_table[ATEXIT_MAX];
2920005 //----------------------------------------------------------------------
2920006 void
2920007 _atexit_setup (void)
2920008 {
2920009     int a;
2920010     //
2920011     for (a = 0; a < ATEXIT_MAX; a++)
2920012       {
2920013         _atexit_table[a] = NULL;
2920014       }
2920015 }
2920016 //----------------------------------------------------------------------
2920017 int
2920018 atexit (atexit_t function)
2920019 {
2920020     int a;
2920021     //
```

```
2920022     if (function == NULL)
2920023       {
2920024         return (-1);
2920025       }
2920026     //
2920027     for (a = 0; a < ATEXIT_MAX; a++)
2920028       {
2920029         if (_atexit_table[a] == NULL)
2920030           {
2920031             _atexit_table[a] = function;
2920032             return (0);
2920033           }
2920034       }
2920035     //
2920036     return (-1);
2920037 }
```

101.10.6 lib/stdlib/atoi.c

Si veda la sezione 94.5.

```
2930001 #include <stdlib.h>
2930002 #include <ctype.h>
2930003 //----------------------------------------------------------------------
2930004 int
2930005 atoi (const char *string)
2930006 {
2930007     int i;
2930008     int sign = +1;
2930009     int number;
2930010     //
2930011     for (i = 0; isspace (string[i]); i++)
2930012       {
2930013         ;
2930014       }
2930015     //
2930016     if (string[i] == '+')
2930017       {
2930018         sign = +1;
2930019         i++;
2930020       }
2930021     else if (string[i] == '-')
2930022       {
2930023         sign = -1;
2930024         i++;
2930025       }
2930026     //
2930027     for (number = 0; isdigit (string[i]); i++)
```

```
2930028         {
2930029           number *= 10;
2930030           number += (string[i] - '0');
2930031         }
2930032       //
2930033       number *= sign;
2930034       //
2930035       return number;
2930036   }
```

101.10.7 lib/stdlib/atol.c

«

Si veda la sezione 94.5.

```
2940001   #include <stdlib.h>
2940002   #include <ctype.h>
2940003   //----------------------------------------------------------------------
2940004   long int
2940005   atol (const char *string)
2940006   {
2940007       int      i;
2940008       int      sign = +1;
2940009       long int number;
2940010       //
2940011       for (i = 0; isspace (string[i]); i++)
2940012         {
2940013           ;
2940014         }
2940015       //
2940016       if (string[i] == '+')
2940017         {
2940018           sign = +1;
2940019           i++;
2940020         }
2940021       else if (string[i] == '-')
2940022         {
2940023           sign = -1;
2940024           i++;
2940025         }
2940026       //
2940027       for (number = 0; isdigit (string[i]); i++)
2940028         {
2940029           number *= 10;
2940030           number += (string[i] - '0');
2940031         }
2940032       //
2940033       number *= sign;
2940034       //
```

```
2940035     return number;
2940036 }
```

101.10.8 lib/stdlib/div.c

«

Si veda la sezione 94.15.

```
2950001 #include <stdlib.h>
2950002 //----------------------------------------------------------------------
2950003 div_t
2950004 div (int numer, int denom)
2950005 {
2950006     div_t d;
2950007     d.quot = numer / denom;
2950008     d.rem = numer % denom;
2950009     return d;
2950010 }
```

101.10.9 lib/stdlib/environment.c

«

Si veda la sezione 97.1.

```
2960001 #include <stdlib.h>
2960002 #include <string.h>
2960003 //----------------------------------------------------------------------
2960004 // This file contains a non standard definition, related to the
2960005 // environment handling.
2960006 //
2960007 // The file 'crt0.s', before calling the main function, calls the
2960008 // function '_environment_setup()', that is responsible for initializing
2960009 // the array '_environment_table[][]' and for copying the content
2960010 // of the environment, as it comes from the 'exec()' system call.
2960011 //
2960012 // The pointers to the environment strings organised inside the
2960013 // array '_environment_table[][]', are also copied inside the
2960014 // array of pointers '_environment[]'.
2960015 //
2960016 // After all that is done, inside 'crt0.s', the pointer to
2960017 // '_environment[]' is copied to the traditional variable 'environ'
2960018 // and also to the previous value of the pointer variable 'envp'.
2960019 //
2960020 // This way, applications will get the environment, but organised
2960021 // inside the table '_environment_table[][]'. So, functions like
2960022 // 'getenv()' and 'setenv()' do know where to look for.
2960023 //
2960024 // It is useful to notice that there is no prototype and no extern
2960025 // declaration inside the file <stdlib.h>, about this function
```

```
2960026 // and these arrays, because applications do not have to know about
2960027 // it.
2960028 //
2960029 // Please notice that 'environ' could be just the same as
2960030 // '_environment' here, but the common use puts 'environ' inside
2960031 // <unistd.h>, although for this implementation it should be better
2960032 // placed inside <stdlib.h>.
2960033 //
2960034 //----------------------------------------------------------------------
2960035 char  _environment_table[ARG_MAX/32][ARG_MAX/16];
2960036 char *_environment[ARG_MAX/32+1];
2960037 //----------------------------------------------------------------------
2960038 void
2960039 _environment_setup (char *envp[])
2960040 {
2960041     int e;
2960042     int s;
2960043     //
2960044     // Reset the '_environment_table[][]' array.
2960045     //
2960046     for (e = 0; e < ARG_MAX/32; e++)
2960047       {
2960048         for (s = 0; s < ARG_MAX/16; s++)
2960049           {
2960050             _environment_table[e][s] = 0;
2960051           }
2960052       }
2960053     //
2960054     // Set the '_environment[]' pointers. The final extra element must
2960055     // be a NULL pointer.
2960056     //
2960057     for (e = 0; e < ARG_MAX/32 ; e++)
2960058       {
2960059         _environment[e] = _environment_table[e];
2960060       }
2960061     _environment[ARG_MAX/32] = NULL;
2960062     //
2960063     // Copy the environment inside the array, but only if 'envp' is
2960064     // not NULL.
2960065     //
2960066     if (envp != NULL)
2960067       {
2960068         for (e = 0; envp[e] != NULL && e < ARG_MAX/32; e++)
2960069           {
2960070             strncpy (_environment_table[e], envp[e], (ARG_MAX/16)-1);
2960071           }
2960072       }
```

```
2960073 }
```

101.10.10 lib/stdlib/exit.c

Si veda la sezione 94.4.

```
2970001 #include <stdlib.h>
2970002 #include <stdio.h>
2970003 //----------------------------------------------------------------------
2970004 extern atexit_t _atexit_table[];
2970005 //----------------------------------------------------------------------
2970006 void
2970007 exit (int status)
2970008 {
2970009     int a;
2970010     //
2970011     // The "at exit" functions must be called in reverse order.
2970012     //
2970013     for (a = (ATEXIT_MAX - 1); a >= 0; a--)
2970014       {
2970015         if (_atexit_table[a] != NULL)
2970016           {
2970017             (*_atexit_table[a]) ();
2970018           }
2970019       }
2970020     //
2970021     // Now: really exit.
2970022     //
2970023     _Exit (status);
2970024     //
2970025     // Should not return from system call, but if it does, loop
2970026     // forever:
2970027     //
2970028     while (1);
2970029 }
```

101.10.11 lib/stdlib/getenv.c

Si veda la sezione 94.51.

```
2980001 #include <stdlib.h>
2980002 #include <string.h>
2980003 //----------------------------------------------------------------------
2980004 extern char *_environment[];
2980005 //----------------------------------------------------------------------
2980006 char *
2980007 getenv (const char *name)
```

```
2980008 {
2980009     int   e;        // First index: environment table items.
2980010     int   f;        // Second index: environment string scan.
2980011     char *value;    // Pointer to the environment value found.
2980012     //
2980013     // Check if the input is valid. No error is reported.
2980014     //
2980015     if (name == NULL || strlen (name) == 0)
2980016       {
2980017         return (NULL);
2980018       }
2980019     //
2980020     // Scan the environment table items, with index 'e'. The pointer
2980021     // 'value' is initialized to NULL. If the pointer 'value' gets a
2980022     // valid  pointer, the environment variable was found and a
2980023     // pointer to the beginning of its value is available.
2980024     //
2980025     for (value = NULL, e = 0; e < ARG_MAX/32; e++)
2980026       {
2980027         //
2980028         // Scan the string of the environment item, with index 'f'.
2980029         // The scan continue until 'name[f]' and '_environment[e][f]'
2980030         // are equal.
2980031         //
2980032         for (f = 0;
2980033              f < ARG_MAX/16-1 && name[f] == _environment[e][f];
2980034              f++)
2980035           {
2980036             ; // Just scan.
2980037           }
2980038         //
2980039         // At this point, 'name[f]' and '_environment[e][f]' are
2980040         // different: if 'name[f]' is zero the name string is
2980041         // terminated; if '_environment[e][f]' is also equal to '=',
2980042         // the environment item is corresponding to the requested name.
2980043         //
2980044         if (name[f] == 0 && _environment[e][f] == '=')
2980045           {
2980046             //
2980047             // The pointer to the beginning of the environment value is
2980048             // calculated, and the external loop exit.
2980049             //
2980050             value = &_environment[e][f+1];
2980051             break;
2980052           }
2980053       }
2980054     //
2980055     // The 'value' is returned: if it is still NULL, then, no
```

```
2980056      // environment variable with the requested name was found.
2980057      //
2980058      return (value);
2980059  }
```

101.10.12 lib/stdlib/labs.c

«

Si veda la sezione 94.3.

```
2990001  #include <stdlib.h>
2990002  //----------------------------------------------------------------------
2990003  long int
2990004  labs (long int j)
2990005  {
2990006      if (j < 0)
2990007        {
2990008          return -j;
2990009        }
2990010      else
2990011        {
2990012          return j;
2990013        }
2990014  }
```

101.10.13 lib/stdlib/ldiv.c

«

Si veda la sezione 94.15.

```
3000001  #include <stdlib.h>
3000002  //----------------------------------------------------------------------
3000003  ldiv_t
3000004  ldiv (long int numer, long int denom)
3000005  {
3000006      ldiv_t d;
3000007      d.quot = numer / denom;
3000008      d.rem = numer % denom;
3000009      return d;
3000010  }
```

101.10.14 lib/stdlib/putenv.c

«

Si veda la sezione 94.82.

```
3010001  #include <stdlib.h>
3010002  #include <string.h>
3010003  #include <errno.h>
```

```
3010004 //----------------------------------------------------------------------
3010005 extern char *_environment[];
3010006 //----------------------------------------------------------------------
3010007 int
3010008 putenv (const char *string)
3010009 {
3010010     int   e;        // First index: environment table items.
3010011     int   f;        // Second index: environment string scan.
3010012     //
3010013     // Check if the input is empty. No error is reported.
3010014     //
3010015     if (string == NULL || strlen (string) == 0)
3010016       {
3010017         return (0);
3010018       }
3010019     //
3010020     // Check if the input is valid: there must be a `=' sign.
3010021     // Error here is reported.
3010022     //
3010023     if (strchr (string, '=') == NULL)
3010024       {
3010025         errset(EINVAL);                 // Invalid argument.
3010026         return (-1);
3010027       }
3010028     //
3010029     // Scan the environment table items, with index `e'. The intent is
3010030     // to find a previous environment variable with the same name.
3010031     //
3010032     for (e = 0; e < ARG_MAX/32; e++)
3010033       {
3010034         //
3010035         // Scan the string of the environment item, with index `f'.
3010036         // The scan continue until `string[f]' and `_environment[e][f]'
3010037         // are equal.
3010038         //
3010039         for (f = 0;
3010040              f < ARG_MAX/16-1 && string[f] == _environment[e][f];
3010041              f++)
3010042           {
3010043             ; // Just scan.
3010044           }
3010045         //
3010046         // At this point, `string[f-1]' and `_environment[e][f-1]'
3010047         // should contain `='. If it is so, the environment is replaced.
3010048         //
3010049         if (string[f-1] == '=' && _environment[e][f-1] == '=')
3010050           {
3010051             //
```

```
3010052                 // The environment item was found: now replace the pointer.
3010053                 //
3010054                 _environment[e] = string;
3010055                 //
3010056                 // Return.
3010057                 //
3010058                 return (0);
3010059               }
3010060           }
3010061       //
3010062       // The item was not found. Scan again for a free slot.
3010063       //
3010064       for (e = 0; e < ARG_MAX/32; e++)
3010065         {
3010066           if (_environment[e] == NULL || _environment[e][0] == 0)
3010067             {
3010068               //
3010069               // An empty item was found and the pointer will be
3010070               // replaced.
3010071               //
3010072               _environment[e] = string;
3010073               //
3010074               // Return.
3010075               //
3010076               return (0);
3010077             }
3010078         }
3010079       //
3010080       // Sorry: the empty slot was not found!
3010081       //
3010082       errset (ENOMEM);             // Not enough space.
3010083       return (-1);
3010084 }
```

101.10.15 lib/stdlib/qsort.c

Si veda la sezione 94.84.

```
3020001 #include <stdlib.h>
3020002 #include <string.h>
3020003 #include <errno.h>
3020004 //----------------------------------------------------------------------
3020005 static int  part (char *array, size_t size, int a, int z,
3020006                   int (*compare)(const void *, const void *));
3020007 static void sort (char *array, size_t size, int a, int z,
3020008                   int (*compare)(const void *, const void *));
3020009 //----------------------------------------------------------------------
3020010 void
```

```
3020011 qsort (void *base, size_t nmemb, size_t size,
3020012        int (*compare)(const void *, const void *))
3020013 {
3020014     if (size <= 1)
3020015       {
3020016         //
3020017         // There is nothing to sort!
3020018         //
3020019         return;
3020020       }
3020021     else
3020022       {
3020023         sort ((char *) base, size, 0, (int) (nmemb - 1), compare);
3020024       }
3020025 }
3020026 //----------------------------------------------------------------------
3020027 static void
3020028 sort (char *array, size_t size, int a, int z,
3020029       int (*compare)(const void *, const void *))
3020030 {
3020031     int loc;
3020032     //
3020033     if (z > a)
3020034       {
3020035         loc = part (array, size, a, z, compare);
3020036         if (loc >= 0)
3020037           {
3020038             sort (array, size, a, loc-1, compare);
3020039             sort (array, size, loc+1, z, compare);
3020040           }
3020041       }
3020042 }
3020043 
3020044 //----------------------------------------------------------------------
3020045 static int
3020046 part (char *array, size_t size, int a, int z,
3020047       int (*compare)(const void *, const void *))
3020048 {
3020049     int   i;
3020050     int   loc;
3020051     char *swap;
3020052     //
3020053     if (z <= a)
3020054       {
3020055         errset (EUNKNOWN);       // Should never happen.
3020056         return (-1);
3020057       }
3020058     //
```

```
3020059       // Index 'i' after the first element; index 'loc' at the last
3020060       // position.
3020061       //
3020062       i   = a + 1;
3020063       loc = z;
3020064       //
3020065       // Prepare space in memory for element swap.
3020066       //
3020067       swap = malloc (size);
3020068       if (swap == NULL)
3020069         {
3020070           errset (ENOMEM);
3020071           return (-1);
3020072         }
3020073       //
3020074       // Loop as long as index 'loc' is higher than index 'i'.
3020075       // When index 'loc' is less or equal to index 'i',
3020076       // then, index 'loc' is the right position for the
3020077       // first element of the current piece of array.
3020078       //
3020079       for (;;)
3020080         {
3020081           //
3020082           // Index 'i' goes up...
3020083           //
3020084           for (;i < loc; i++)
3020085             {
3020086               if  (compare (&array[i*size], &array[a*size]) > 0)
3020087                 {
3020088                   break;
3020089                 }
3020090             }
3020091           //
3020092           // Index 'loc' gose down...
3020093           //
3020094           for (;; loc--)
3020095             {
3020096               if  (compare (&array[loc*size], &array[a*size]) <= 0)
3020097                 {
3020098                   break;
3020099                 }
3020100             }
3020101           //
3020102           // Swap elements related to index 'i' and 'loc'.
3020103           //
3020104           if (loc <= i)
3020105             {
3020106               //
```

```
3020107                // The array is completely scanned.
3020108                //
3020109                break;
3020110              }
3020111            else
3020112              {
3020113                memcpy (swap, &array[loc*size], size);
3020114                memcpy (&array[loc*size], &array[i*size], size);
3020115                memcpy (&array[i*size], swap, size);
3020116              }
3020117          }
3020118        //
3020119        // Swap the first element with the one related to the
3020120        // index 'loc'.
3020121        //
3020122        memcpy (swap, &array[loc*size], size);
3020123        memcpy (&array[loc*size], &array[a*size], size);
3020124        memcpy (&array[a*size], swap, size);
3020125        //
3020126        // Free the swap memory.
3020127        //
3020128        free (swap);
3020129        //
3020130        // Return the index 'loc'.
3020131        //
3020132        return (loc);
3020133    }
3020134
```

101.10.16 lib/stdlib/rand.c

«

Si veda la sezione 94.85.

```
3030001    #include <stdlib.h>
3030002    //----------------------------------------------------------------------
3030003    static unsigned int _srand = 1; // The '_srand' rank must be at least
3030004                                    // 'unsigned int' and must be able to
3030005                                    // represent the value 'RAND_MAX'.
3030006    //----------------------------------------------------------------------
3030007    int
3030008    rand (void)
3030009    {
3030010        _srand = _srand * 12345 + 123;
3030011        return _srand % ((unsigned int) RAND_MAX + 1);
3030012    }
3030013    //----------------------------------------------------------------------
3030014    void
3030015    srand (unsigned int seed)
```

```
3030016 {
3030017     _srand = seed;
3030018 }
```

101.10.17 lib/stdlib/setenv.c

«

Si veda la sezione 94.94.

```
3040001 #include <stdlib.h>
3040002 #include <string.h>
3040003 #include <errno.h>
3040004 //----------------------------------------------------------------------
3040005 extern char *_environment[];
3040006 extern char *_environment_table[];
3040007 //----------------------------------------------------------------------
3040008 int
3040009 setenv (const char *name, const char *value, int overwrite)
3040010 {
3040011     int   e;        // First index: environment table items.
3040012     int   f;        // Second index: environment string scan.
3040013     //
3040014     // Check if the input is empty. No error is reported.
3040015     //
3040016     if (name == NULL || strlen (name) == 0)
3040017       {
3040018         return (0);
3040019       }
3040020     //
3040021     // Check if the input is valid: error here is reported.
3040022     //
3040023     if (strchr (name, '=') != NULL)
3040024       {
3040025         errset(EINVAL);                 // Invalid argument.
3040026         return (-1);
3040027       }
3040028     //
3040029     // Check if the input is too big.
3040030     //
3040031     if ((strlen (name) + strlen (value) + 2) > ARG_MAX/16)
3040032       {
3040033         //
3040034         // The environment to be saved is bigger than the
3040035         // available string size, inside '_environment_table[]'.
3040036         //
3040037         errset (ENOMEM);                // Not enough space.
3040038         return (-1);
3040039       }
3040040     //
```

```
3040041      // Scan the environment table items, with index 'e'. The intent is
3040042      // to find a previous environment variable with the same name.
3040043      //
3040044      for (e = 0; e < ARG_MAX/32; e++)
3040045        {
3040046          //
3040047          // Scan the string of the environment item, with index 'f'.
3040048          // The scan continue until 'name[f]' and '_environment[e][f]'
3040049          // are equal.
3040050          //
3040051          for (f = 0;
3040052               f < ARG_MAX/16-1 && name[f] == _environment[e][f];
3040053               f++)
3040054            {
3040055              ; // Just scan.
3040056            }
3040057          //
3040058          // At this point, 'name[f]' and '_environment[e][f]' are
3040059          // different: if 'name[f]' is zero the name string is
3040060          // terminated; if '_environment[e][f]' is also equal to '=',
3040061          // the environment item is corresponding to the requested name.
3040062          //
3040063          if (name[f] == 0 && _environment[e][f] == '=')
3040064            {
3040065              //
3040066              // The environment item was found; if it can be overwritten,
3040067              // the write is done.
3040068              //
3040069              if (overwrite)
3040070                {
3040071                  //
3040072                  // To be able to handle both 'setenv()' and 'putenv()',
3040073                  // before removing the item, it is fixed the pointer to
3040074                  // the global environment table.
3040075                  //
3040076                  _environment[e] = _environment_table[e];
3040077                  //
3040078                  // Now copy the new environment. The string size was
3040079                  // already checked.
3040080                  //
3040081                  strcpy (_environment[e], name);
3040082                  strcat (_environment[e], "=");
3040083                  strcat (_environment[e], value);
3040084                  //
3040085                  // Return.
3040086                  //
3040087                  return (0);
3040088                }
```

```
3040089                 //
3040090                 // Cannot overwrite!
3040091                 //
3040092                 errset (EUNKNOWN);
3040093                 return (-1);
3040094               }
3040095           }
3040096         //
3040097         // The item was not found. Scan again for a free slot.
3040098         //
3040099         for (e = 0; e < ARG_MAX/32; e++)
3040100           {
3040101             if (_environment[e] == NULL || _environment[e][0] == 0)
3040102               {
3040103                 //
3040104                 // An empty item was found. To be able to handle both
3040105                 // `setenv()' and `putenv()', it is fixed the pointer to
3040106                 // the global environment table.
3040107                 //
3040108                 _environment[e] = _environment_table[e];
3040109                 //
3040110                 // Now copy the new environment. The string size was
3040111                 // already checked.
3040112                 //
3040113                 strcpy (_environment[e], name);
3040114                 strcat (_environment[e], "=");
3040115                 strcat (_environment[e], value);
3040116                 //
3040117                 // Return.
3040118                 //
3040119                 return (0);
3040120               }
3040121           }
3040122         //
3040123         // Sorry: the empty slot was not found!
3040124         //
3040125         errset (ENOMEM);            // Not enough space.
3040126         return (-1);
3040127   }
```

101.10.18 lib/stdlib/strtol.c

«

Si veda la sezione 94.121.

```
3050001   #include <stdlib.h>
3050002   #include <ctype.h>
3050003   #include <errno.h>
3050004   #include <limits.h>
```

```
3050005 #include <stdbool.h>
3050006 //----------------------------------------------------------------------
3050007 #define isoctal(C)  ((int) (C >= '0' && C <= '7'))
3050008 //----------------------------------------------------------------------
3050009 long int
3050010 strtol (const char *restrict string, char **restrict endptr, int base)
3050011 {
3050012     int      i;
3050013     int      sign = +1;
3050014     long int number;
3050015     long int previous;
3050016     int      digit;
3050017     //
3050018     bool     flag_prefix_oct = 0;
3050019     bool     flag_prefix_exa = 0;
3050020     bool     flag_prefix_dec = 0;
3050021     //
3050022     // Check base and string.
3050023     //
3050024     if (base < 0
3050025      || base > 36
3050026      || base == 1               // With base 1 cannot do anything.
3050027      || string == NULL
3050028      || string[0] == 0)
3050029       {
3050030         if (endptr != NULL) *endptr = string;
3050031         errset (EINVAL);                 // Invalid argument.
3050032         return ((long int) 0);
3050033       }
3050034     //
3050035     // Eat initial spaces.
3050036     //
3050037     for (i = 0; isspace (string[i]); i++)
3050038       {
3050039         ;
3050040       }
3050041     //
3050042     // Check sign.
3050043     //
3050044     if (string[i] == '+')
3050045       {
3050046         sign = +1;
3050047         i++;
3050048       }
3050049     else if (string[i] == '-')
3050050       {
3050051         sign = -1;
3050052         i++;
```

```
3050053         }
3050054       //
3050055       // Check for prefix.
3050056       //
3050057       if (string[i] == '0')
3050058         {
3050059           if (string[i+1] == 'x' || string[i+1] == 'X')
3050060             {
3050061               flag_prefix_exa = 1;
3050062             }
3050063           else if (isoctal (string[i+1]))
3050064             {
3050065               flag_prefix_oct = 1;
3050066             }
3050067           else
3050068             {
3050069               flag_prefix_dec = 1;
3050070             }
3050071         }
3050072       else if (isdigit (string[i]))
3050073         {
3050074           flag_prefix_dec = 1;
3050075         }
3050076       //
3050077       // Check compatibility with requested base.
3050078       //
3050079       if (flag_prefix_exa)
3050080         {
3050081           //
3050082           // At the moment, there is a zero and a 'x'. Might be
3050083           // exadecimal, or might be a number base 33 or more.
3050084           //
3050085           if (base == 0)
3050086             {
3050087               base = 16;
3050088             }
3050089           else if (base == 16)
3050090             {
3050091               ;   // Ok.
3050092             }
3050093           else if (base >= 33)
3050094             {
3050095               ;   // Ok.
3050096             }
3050097           else
3050098             {
3050099               //
3050100               // Incompatible sequence: only the initial zero is reported.
```

```
3050101                 //
3050102                 if (endptr != NULL) *endptr = &string[i+1];
3050103                 return ((long int) 0);
3050104               }
3050105             //
3050106             // Move on, after the ‘0x’ prefix.
3050107             //
3050108             i += 2;
3050109           }
3050110         //
3050111         if (flag_prefix_oct)
3050112           {
3050113             //
3050114             // There is a zero and a digit.
3050115             //
3050116             if (base == 0)
3050117               {
3050118                 base = 8;
3050119               }
3050120             //
3050121             // Move on, after the ‘0’ prefix.
3050122             //
3050123             i += 1;
3050124           }
3050125         //
3050126         if (flag_prefix_dec)
3050127           {
3050128             if (base == 0)
3050129               {
3050130                 base = 10;
3050131               }
3050132           }
3050133         //
3050134         // Scan the string.
3050135         //
3050136         for (number = 0; string[i] != 0; i++)
3050137           {
3050138             if      (string[i] >= '0' && string[i] <= '9')
3050139               {
3050140                 digit = string[i] - '0';
3050141               }
3050142             else if (string[i] >= 'A' && string[i] <= 'Z')
3050143               {
3050144                 digit = string[i] - 'A' + 10;
3050145               }
3050146             else if (string[i] >= 'a' && string[i] <= 'z')
3050147               {
3050148                 digit = string[i] - 'a' + 10;
```

```
3050149             }
3050150           else
3050151             {
3050152               //
3050153               // This is an out of range digit.
3050154               //
3050155               digit = 999;
3050156             }
3050157           //
3050158           // Give a sign to the digit.
3050159           //
3050160           digit *= sign;
3050161           //
3050162           // Compare with the base.
3050163           //
3050164           if (base > (digit * sign))
3050165             {
3050166               //
3050167               // Check if the current digit can be safely computed.
3050168               //
3050169               previous = number;
3050170               number  *= base;
3050171               number  += digit;
3050172               if (number / base != previous)
3050173                 {
3050174                   //
3050175                   // Out of range.
3050176                   //
3050177                   if (endptr != NULL) *endptr = &string[i+1];
3050178                   errset (ERANGE);         // Result too large.
3050179                   if (sign > 0)
3050180                     {
3050181                       return (LONG_MAX);
3050182                     }
3050183                   else
3050184                     {
3050185                       return (LONG_MIN);
3050186                     }
3050187                 }
3050188             }
3050189           else
3050190             {
3050191               if (endptr != NULL) *endptr = &string[i];
3050192               return (number);
3050193             }
3050194         }
3050195       //
3050196       // The string is finished.
```

```
3050197     //
3050198     if (endptr != NULL) *endptr = &string[i];
3050199     //
3050200     return (number);
3050201 }
```

101.10.19 lib/stdlib/strtoul.c

«

Si veda la sezione 94.121.

```
3060001 #include <stdlib.h>
3060002 #include <ctype.h>
3060003 #include <errno.h>
3060004 #include <limits.h>
3060005 //----------------------------------------------------------------------
3060006 // A really poor implementation. ,-(
3060007 //
3060008 unsigned long int
3060009 strtoul (const char *restrict string, char **restrict endptr, int base)
3060010 {
3060011     return ((unsigned long int) strtol (string, endptr, base));
3060012 }
```

101.10.20 lib/stdlib/unsetenv.c

«

Si veda la sezione 94.94.

```
3070001 #include <stdlib.h>
3070002 #include <string.h>
3070003 #include <errno.h>
3070004 //----------------------------------------------------------------------
3070005 extern char *_environment[];
3070006 extern char *_environment_table[];
3070007 //----------------------------------------------------------------------
3070008 int
3070009 unsetenv (const char *name)
3070010 {
3070011     int     e;          // First index: environment table items.
3070012     int     f;          // Second index: environment string scan.
3070013     //
3070014     // Check if the input is empty. No error is reported.
3070015     //
3070016     if (name == NULL || strlen (name) == 0)
3070017       {
3070018         return (0);
3070019       }
3070020     //
```

```
3070021     // Check if the input is valid: error here is reported.
3070022     //
3070023     if (strchr (name, '=') != NULL)
3070024       {
3070025         errset (EINVAL);                  // Invalid argument.
3070026         return (-1);
3070027       }
3070028     //
3070029     // Scan the environment table items, with index 'e'.
3070030     //
3070031     for (e = 0; e < ARG_MAX/32; e++)
3070032       {
3070033         //
3070034         // Scan the string of the environment item, with index 'f'.
3070035         // The scan continue until 'name[f]' and '_environment[e][f]'
3070036         // are equal.
3070037         //
3070038         for (f = 0;
3070039              f < ARG_MAX/16-1 && name[f] == _environment[e][f];
3070040              f++)
3070041           {
3070042             ; // Just scan.
3070043           }
3070044         //
3070045         // At this point, 'name[f]' and '_environment[e][f]' are
3070046         // different: if 'name[f]' is zero the name string is
3070047         // terminated; if '_environment[e][f]' is also equal to '=',
3070048         // the environment item is corresponding to the requested name.
3070049         //
3070050         if (name[f] == 0 && _environment[e][f] == '=')
3070051           {
3070052             //
3070053             // The environment item was found and it have to be removed.
3070054             // To be able to handle both 'setenv()' and 'putenv()',
3070055             // before removing the item, it is fixed the pointer to
3070056             // the global environment table.
3070057             //
3070058             _environment[e] = _environment_table[e];
3070059             //
3070060             // Now remove the environment item.
3070061             //
3070062             _environment[e][0] = 0;
3070063             break;
3070064           }
3070065       }
3070066     //
3070067     // Work done fine.
3070068     //
```

```
3070069     return (0);
3070070 }
```

101.11 os16: «lib/string.h»

«

Si veda la sezione 97.2.

```
3080001 #ifndef _STRING_H
3080002 #define _STRING_H       1
3080003
3080004 #include <const.h>
3080005 #include <restrict.h>
3080006 #include <const.h>
3080007 #include <size_t.h>
3080008 #include <NULL.h>
3080009 //----------------------------------------------------------------------
3080010 void   *memccpy (void *restrict dst, const void *restrict org, int c,
3080011                  size_t n);
3080012 void   *memchr  (const void *memory, int c, size_t n);
3080013 int     memcmp  (const void *memory1, const void *memory2, size_t n);
3080014 void   *memcpy  (void *restrict dst, const void *restrict org, size_t n);
3080015 void   *memmove (void *dst, const void *org, size_t n);
3080016 void   *memset  (void *memory, int c, size_t n);
3080017 char   *strcat  (char *restrict dst, const char *restrict org);
3080018 char   *strchr  (const char *string, int c);
3080019 int     strcmp  (const char *string1, const char *string2);
3080020 int     strcoll (const char *string1, const char *string2);
3080021 char   *strcpy  (char *restrict dst, const char *restrict org);
3080022 size_t strcspn (const char *string, const char *reject);
3080023 char   *strdup  (const char *string);
3080024 char   *strerror (int errnum);
3080025 size_t strlen  (const char *string);
3080026 char   *strncat (char *restrict dst, const char *restrict org, size_t n);
3080027 int     strncmp (const char *string1, const char *string2, size_t n);
3080028 char   *strncpy (char *restrict dst, const char *restrict org, size_t n);
3080029 char   *strpbrk (const char *string, const char *accept);
3080030 char   *strrchr (const char *string, int c);
3080031 size_t strspn  (const char *string, const char *accept);
3080032 char   *strstr  (const char *string, const char *substring);
3080033 char   *strtok  (char *restrict string, const char *restrict delim);
3080034 size_t strxfrm (char *restrict dst, const char *restrict org, size_t n);
3080035 //----------------------------------------------------------------------
3080036
3080037
3080038
3080039 #endif
```

101.11.1 lib/string/memccpy.c

Si veda la sezione 94.67.

```
3090001 #include <string.h>
3090002 //----------------------------------------------------------------------
3090003 void *
3090004 memccpy (void *restrict dst, const void *restrict org, int c, size_t n)
3090005 {
3090006     char *d = (char *) dst;
3090007     char *o = (char *) org;
3090008     size_t i;
3090009     for (i = 0; n > 0 && i < n; i++)
3090010       {
3090011         d[i] = o[i];
3090012         if (d[i] == (char) c)
3090013           {
3090014             return ((void *) &d[i+1]);
3090015           }
3090016       }
3090017     return (NULL);
3090018 }
```

101.11.2 lib/string/memchr.c

Si veda la sezione 94.68.

```
3100001 #include <string.h>
3100002 //----------------------------------------------------------------------
3100003 void *
3100004 memchr (const void *memory, int c, size_t n)
3100005 {
3100006     char *m = (char *) memory;
3100007     size_t i;
3100008     for (i = 0; n > 0 && i < n; i++)
3100009       {
3100010         if (m[i] == (char) c)
3100011           {
3100012             return (void *) (m + i);
3100013           }
3100014       }
3100015     return NULL;
3100016 }
```

101.11.3 lib/string/memcmp.c

«

Si veda la sezione 94.69.

```
3110001 #include <string.h>
3110002 //----------------------------------------------------------------------
3110003 int
3110004 memcmp (const void *memory1, const void *memory2, size_t n)
3110005 {
3110006     char *a = (char *) memory1;
3110007     char *b = (char *) memory2;
3110008     size_t i;
3110009     for (i = 0; n > 0 && i < n; i++)
3110010       {
3110011         if (a[i] > b[i])
3110012           {
3110013             return 1;
3110014           }
3110015         else if (a[i] < b[i])
3110016           {
3110017             return -1;
3110018           }
3110019       }
3110020     return 0;
3110021 }
```

101.11.4 lib/string/memcpy.c

«

Si veda la sezione 94.70.

```
3120001 #include <string.h>
3120002 //----------------------------------------------------------------------
3120003 void *
3120004 memcpy (void *restrict dst, const void *restrict org, size_t n)
3120005 {
3120006     char *d = (char *) dst;
3120007     char *o = (char *) org;
3120008     size_t i;
3120009     for (i = 0; n > 0 && i < n; i++)
3120010       {
3120011         d[i] = o[i];
3120012       }
3120013     return dst;
3120014 }
```

101.11.5 lib/string/memmove.c

«

Si veda la sezione 94.71.

```
3130001 #include <string.h>
3130002 //----------------------------------------------------------------------
3130003 void *
3130004 memmove (void *dst, const void *org, size_t n)
3130005 {
3130006     char *d = (char *) dst;
3130007     char *o = (char *) org;
3130008     size_t i;
3130009     //
3130010     // Depending on the memory start locations, copy may be direct or
3130011     // reverse, to avoid overwriting before the relocation is done.
3130012     //
3130013     if (d < o)
3130014       {
3130015         for (i = 0; i < n; i++)
3130016           {
3130017             d[i] = o[i];
3130018           }
3130019       }
3130020     else if (d == o)
3130021       {
3130022         //
3130023         // Memory locations are already the same.
3130024         //
3130025         ;
3130026       }
3130027     else
3130028       {
3130029         for (i = n - 1; i >= 0; i--)
3130030           {
3130031             d[i] = o[i];
3130032           }
3130033       }
3130034     return dst;
3130035 }
```

101.11.6 lib/string/memset.c

«

Si veda la sezione 94.72.

```
3140001 #include <string.h>
3140002 //----------------------------------------------------------------------
3140003 void *
3140004 memset (void *memory, int c, size_t n)
3140005 {
```

```
3140006     char  *m = (char *) memory;
3140007     size_t i;
3140008     for (i = 0; n > 0 && i < n; i++)
3140009       {
3140010         m[i] = (char) c;
3140011       }
3140012     return memory;
3140013 }
```

101.11.7 lib/string/strcat.c

«

Si veda la sezione 94.104.

```
3150001 #include <string.h>
3150002 //----------------------------------------------------------------------
3150003 char *
3150004 strcat (char *restrict dst, const char *restrict org)
3150005 {
3150006     size_t i;
3150007     size_t j;
3150008     for (i = 0; dst[i] != 0; i++)
3150009       {
3150010         ; // Just look for the null character.
3150011       }
3150012     for (j = 0; org[j] != 0; i++, j++)
3150013       {
3150014         dst[i] = org[j];
3150015       }
3150016     dst[i] = 0;
3150017     return dst;
3150018 }
```

101.11.8 lib/string/strchr.c

«

Si veda la sezione 94.105.

```
3160001 #include <string.h>
3160002 //----------------------------------------------------------------------
3160003 char *
3160004 strchr (const char *string, int c)
3160005 {
3160006     size_t i;
3160007     for (i = 0; ; i++)
3160008       {
3160009         if (string[i] == (char) c)
3160010           {
3160011             return (char *) (string + i);
```

```
3160012             }
3160013           else if (string[i] == 0)
3160014             {
3160015               return NULL;
3160016             }
3160017         }
3160018 }
```

101.11.9 lib/string/strcmp.c

Si veda la sezione 94.106.

```
3170001 #include <string.h>
3170002 //----------------------------------------------------------------------
3170003 int
3170004 strcmp (const char *string1, const char *string2)
3170005 {
3170006     char *a = (char *) string1;
3170007     char *b = (char *) string2;
3170008     size_t i;
3170009     for (i = 0; ; i++)
3170010       {
3170011         if (a[i] > b[i])
3170012           {
3170013             return 1;
3170014           }
3170015         else if (a[i] < b[i])
3170016           {
3170017             return -1;
3170018           }
3170019         else if (a[i] == 0 && b[i] == 0)
3170020           {
3170021             return 0;
3170022           }
3170023       }
3170024 }
```

101.11.10 lib/string/strcoll.c

Si veda la sezione 94.106.

```
3180001 #include <string.h>
3180002 //----------------------------------------------------------------------
3180003 int
3180004 strcoll (const char *string1, const char *string2)
3180005 {
3180006     return (strcmp (string1, string2));
```

```
3180007 }
```

101.11.11 lib/string/strcpy.c

«

Si veda la sezione 94.108.

```
3190001 #include <string.h>
3190002 //----------------------------------------------------------------------
3190003 char *
3190004 strcpy (char *restrict dst, const char *restrict org)
3190005 {
3190006     size_t i;
3190007     for (i = 0; org[i] != 0; i++)
3190008       {
3190009         dst[i] = org[i];
3190010       }
3190011     dst[i] = 0;
3190012     return dst;
3190013 }
```

101.11.12 lib/string/strcspn.c

«

Si veda la sezione 94.118.

```
3200001 #include <string.h>
3200002 //----------------------------------------------------------------------
3200003 size_t
3200004 strcspn (const char *string, const char *reject)
3200005 {
3200006     size_t i;
3200007     size_t j;
3200008     int found;
3200009     for (i = 0; string[i] != 0; i++)
3200010       {
3200011         for (j = 0, found = 0; reject[j] != 0 || found; j++)
3200012           {
3200013             if (string[i] == reject[j])
3200014               {
3200015                 found = 1;
3200016                 break;
3200017               }
3200018           }
3200019         if (found)
3200020           {
3200021             break;
3200022           }
3200023       }
```

```
3200024     return i;
3200025 }
```

101.11.13 lib/string/strdup.c

Si veda la sezione 94.110.

```
3210001 #include <string.h>
3210002 #include <stdlib.h>
3210003 #include <errno.h>
3210004 //----------------------------------------------------------------------
3210005 char *
3210006 strdup  (const char *string)
3210007 {
3210008     size_t size;
3210009     char  *copy;
3210010     //
3210011     // Get string size: must be added 1, to count the termination null
3210012     // character.
3210013     //
3210014     size = strlen (string) + 1;
3210015     //
3210016     copy = malloc (size);
3210017     //
3210018     if (copy == NULL)
3210019       {
3210020         errset (ENOMEM);         // Not enough memory.
3210021         return (NULL);
3210022       }
3210023     //
3210024     strcpy (copy, string);
3210025     //
3210026     return (copy);
3210027 }
```

101.11.14 lib/string/strerror.c

Si veda la sezione 94.111.

```
3220001 #include <string.h>
3220002 #include <errno.h>
3220003 //----------------------------------------------------------------------
3220004 #define ERROR_MAX 100
3220005 //----------------------------------------------------------------------
3220006 char *
3220007 strerror (int errnum)
3220008 {
```

```
3220009     static char *err[ERROR_MAX];
3220010     //
3220011     err[0]                                  = "No error";
3220012     err[E2BIG]                              = TEXT_E2BIG;
3220013     err[EACCES]                             = TEXT_EACCES;
3220014     err[EADDRINUSE]                         = TEXT_EADDRINUSE;
3220015     err[EADDRNOTAVAIL]                      = TEXT_EADDRNOTAVAIL;
3220016     err[EAFNOSUPPORT]                       = TEXT_EAFNOSUPPORT;
3220017     err[EAGAIN]                             = TEXT_EAGAIN;
3220018     err[EALREADY]                           = TEXT_EALREADY;
3220019     err[EBADF]                              = TEXT_EBADF;
3220020     err[EBADMSG]                            = TEXT_EBADMSG;
3220021     err[EBUSY]                              = TEXT_EBUSY;
3220022     err[ECANCELED]                          = TEXT_ECANCELED;
3220023     err[ECHILD]                             = TEXT_ECHILD;
3220024     err[ECONNABORTED]                       = TEXT_ECONNABORTED;
3220025     err[ECONNREFUSED]                       = TEXT_ECONNREFUSED;
3220026     err[ECONNRESET]                         = TEXT_ECONNRESET;
3220027     err[EDEADLK]                            = TEXT_EDEADLK;
3220028     err[EDESTADDRREQ]                       = TEXT_EDESTADDRREQ;
3220029     err[EDOM]                               = TEXT_EDOM;
3220030     err[EDQUOT]                             = TEXT_EDQUOT;
3220031     err[EEXIST]                             = TEXT_EEXIST;
3220032     err[EFAULT]                             = TEXT_EFAULT;
3220033     err[EFBIG]                              = TEXT_EFBIG;
3220034     err[EHOSTUNREACH]                       = TEXT_EHOSTUNREACH;
3220035     err[EIDRM]                              = TEXT_EIDRM;
3220036     err[EILSEQ]                             = TEXT_EILSEQ;
3220037     err[EINPROGRESS]                        = TEXT_EINPROGRESS;
3220038     err[EINTR]                              = TEXT_EINTR;
3220039     err[EINVAL]                             = TEXT_EINVAL;
3220040     err[EIO]                                = TEXT_EIO;
3220041     err[EISCONN]                            = TEXT_EISCONN;
3220042     err[EISDIR]                             = TEXT_EISDIR;
3220043     err[ELOOP]                              = TEXT_ELOOP;
3220044     err[EMFILE]                             = TEXT_EMFILE;
3220045     err[EMLINK]                             = TEXT_EMLINK;
3220046     err[EMSGSIZE]                           = TEXT_EMSGSIZE;
3220047     err[EMULTIHOP]                          = TEXT_EMULTIHOP;
3220048     err[ENAMETOOLONG]                       = TEXT_ENAMETOOLONG;
3220049     err[ENETDOWN]                           = TEXT_ENETDOWN;
3220050     err[ENETRESET]                          = TEXT_ENETRESET;
3220051     err[ENETUNREACH]                        = TEXT_ENETUNREACH;
3220052     err[ENFILE]                             = TEXT_ENFILE;
3220053     err[ENOBUFS]                            = TEXT_ENOBUFS;
3220054     err[ENODATA]                            = TEXT_ENODATA;
3220055     err[ENODEV]                             = TEXT_ENODEV;
3220056     err[ENOENT]                             = TEXT_ENOENT;
```

```
3220057     err[ENOEXEC]                      = TEXT_ENOEXEC;
3220058     err[ENOLCK]                       = TEXT_ENOLCK;
3220059     err[ENOLINK]                      = TEXT_ENOLINK;
3220060     err[ENOMEM]                       = TEXT_ENOMEM;
3220061     err[ENOMSG]                       = TEXT_ENOMSG;
3220062     err[ENOPROTOOPT]                  = TEXT_ENOPROTOOPT;
3220063     err[ENOSPC]                       = TEXT_ENOSPC;
3220064     err[ENOSR]                        = TEXT_ENOSR;
3220065     err[ENOSTR]                       = TEXT_ENOSTR;
3220066     err[ENOSYS]                       = TEXT_ENOSYS;
3220067     err[ENOTCONN]                     = TEXT_ENOTCONN;
3220068     err[ENOTDIR]                      = TEXT_ENOTDIR;
3220069     err[ENOTEMPTY]                    = TEXT_ENOTEMPTY;
3220070     err[ENOTSOCK]                     = TEXT_ENOTSOCK;
3220071     err[ENOTSUP]                      = TEXT_ENOTSUP;
3220072     err[ENOTTY]                       = TEXT_ENOTTY;
3220073     err[ENXIO]                        = TEXT_ENXIO;
3220074     err[EOPNOTSUPP]                   = TEXT_EOPNOTSUPP;
3220075     err[EOVERFLOW]                    = TEXT_EOVERFLOW;
3220076     err[EPERM]                        = TEXT_EPERM;
3220077     err[EPIPE]                        = TEXT_EPIPE;
3220078     err[EPROTO]                       = TEXT_EPROTO;
3220079     err[EPROTONOSUPPORT]              = TEXT_EPROTONOSUPPORT;
3220080     err[EPROTOTYPE]                   = TEXT_EPROTOTYPE;
3220081     err[ERANGE]                       = TEXT_ERANGE;
3220082     err[EROFS]                        = TEXT_EROFS;
3220083     err[ESPIPE]                       = TEXT_ESPIPE;
3220084     err[ESRCH]                        = TEXT_ESRCH;
3220085     err[ESTALE]                       = TEXT_ESTALE;
3220086     err[ETIME]                        = TEXT_ETIME;
3220087     err[ETIMEDOUT]                    = TEXT_ETIMEDOUT;
3220088     err[ETXTBSY]                      = TEXT_ETXTBSY;
3220089     err[EWOULDBLOCK]                  = TEXT_EWOULDBLOCK;
3220090     err[EXDEV]                        = TEXT_EXDEV;
3220091     err[E_FILE_TYPE]                  = TEXT_E_FILE_TYPE;
3220092     err[E_ROOT_INODE_NOT_CACHED]      = TEXT_E_ROOT_INODE_NOT_CACHED;
3220093     err[E_CANNOT_READ_SUPERBLOCK]     = TEXT_E_CANNOT_READ_SUPERBLOCK;
3220094     err[E_MAP_INODE_TOO_BIG]          = TEXT_E_MAP_INODE_TOO_BIG;
3220095     err[E_MAP_ZONE_TOO_BIG]           = TEXT_E_MAP_ZONE_TOO_BIG;
3220096     err[E_DATA_ZONE_TOO_BIG]          = TEXT_E_DATA_ZONE_TOO_BIG;
3220097     err[E_CANNOT_FIND_ROOT_DEVICE]    = TEXT_E_CANNOT_FIND_ROOT_DEVICE;
3220098     err[E_CANNOT_FIND_ROOT_INODE]     = TEXT_E_CANNOT_FIND_ROOT_INODE;
3220099     err[E_FILE_TYPE_UNSUPPORTED]      = TEXT_E_FILE_TYPE_UNSUPPORTED;
3220100     err[E_ENV_TOO_BIG]                = TEXT_E_ENV_TOO_BIG;
3220101     err[E_LIMIT]                      = TEXT_E_LIMIT;
3220102     err[E_NOT_MOUNTED]                = TEXT_E_NOT_MOUNTED;
3220103     err[E_NOT_IMPLEMENTED]            = TEXT_E_NOT_IMPLEMENTED;
3220104     //
```

```
3220105     if (errnum >= ERROR_MAX || errnum < 0)
3220106       {
3220107         return ("Unknown error");
3220108       }
3220109     //
3220110     return (err[errnum]);
3220111 }
```

101.11.15 lib/string/strlen.c

«

Si veda la sezione 94.112.

```
3230001 #include <string.h>
3230002 //----------------------------------------------------------------------
3230003 size_t
3230004 strlen (const char *string)
3230005 {
3230006     size_t i;
3230007     for (i = 0; string[i] != 0 ; i++)
3230008       {
3230009         ; // Just count.
3230010       }
3230011     return i;
3230012 }
```

101.11.16 lib/string/strncat.c

«

Si veda la sezione 94.104.

```
3240001 #include <string.h>
3240002 //----------------------------------------------------------------------
3240003 char *
3240004 strncat (char *restrict dst, const char *restrict org, size_t n)
3240005 {
3240006     size_t i;
3240007     size_t j;
3240008     for (i = 0; n > 0 && dst[i] != 0; i++)
3240009       {
3240010         ; // Just seek the null character.
3240011       }
3240012     for (j = 0; n > 0 && j < n && org[j] != 0; i++, j++)
3240013       {
3240014         dst[i] = org[j];
3240015       }
3240016     dst[i] = 0;
3240017     return dst;
3240018 }
```

101.11.17 lib/string/strncmp.c

Si veda la sezione 94.106.

```
3250001 #include <string.h>
3250002 //----------------------------------------------------------------------
3250003 int
3250004 strncmp (const char *string1, const char *string2, size_t n)
3250005 {
3250006     size_t i;
3250007     for (i = 0; i < n ; i++)
3250008       {
3250009         if (string1[i] > string2[i])
3250010           {
3250011             return 1;
3250012           }
3250013         else if (string1[i] < string2[i])
3250014           {
3250015             return -1;
3250016           }
3250017         else if (string1[i] == 0 && string2[i] == 0)
3250018           {
3250019             return 0;
3250020           }
3250021       }
3250022     return 0;
3250023 }
```

101.11.18 lib/string/strncpy.c

Si veda la sezione 94.108.

```
3260001 #include <string.h>
3260002 //----------------------------------------------------------------------
3260003 char *
3260004 strncpy (char *restrict dst, const char *restrict org, size_t n)
3260005 {
3260006     size_t i;
3260007     for (i = 0; n > 0 && i < n && org[i] != 0; i++)
3260008       {
3260009         dst[i] = org[i];
3260010       }
3260011     for ( ; n > 0 && i < n; i++)
3260012       {
3260013         dst[i] = 0;
3260014       }
3260015     return dst;
3260016 }
```

101.11.19 lib/string/strpbrk.c

«

Si veda la sezione 94.116.

```
3270001 #include <string.h>
3270002 //----------------------------------------------------------------------
3270003 char *
3270004 strpbrk (const char *string, const char *accept)
3270005 {
3270006     size_t i;
3270007     size_t j;
3270008     for (i = 0; string[i] != 0; i++)
3270009       {
3270010         for (j = 0; accept[j] != 0; j++)
3270011           {
3270012             if (string[i] == accept[j])
3270013               {
3270014                 return (string + i);
3270015               }
3270016           }
3270017       }
3270018     return NULL;
3270019 }
```

101.11.20 lib/string/strrchr.c

«

Si veda la sezione 94.105.

```
3280001 #include <string.h>
3280002 //----------------------------------------------------------------------
3280003 char *
3280004 strrchr (const char *string, int c)
3280005 {
3280006     int i;
3280007     for (i = strlen (string); i >= 0; i--)
3280008       {
3280009         if (string[i] == (char) c)
3280010           {
3280011             break;
3280012           }
3280013       }
3280014     if (i < 0)
3280015       {
3280016         return NULL;
3280017       }
3280018     else
3280019       {
3280020         return (string + i);
3280021       }
```

```
3280022 }
```

101.11.21 lib/string/strspn.c

Si veda la sezione 94.118.

```
3290001 #include <string.h>
3290002 //----------------------------------------------------------------------
3290003 size_t
3290004 strspn (const char *string, const char *accept)
3290005 {
3290006     size_t i;
3290007     size_t j;
3290008     int found;
3290009     for (i = 0; string[i] != 0; i++)
3290010       {
3290011         for (j = 0, found = 0; accept[j] != 0; j++)
3290012           {
3290013             if (string[i] == accept[j])
3290014               {
3290015                 found = 1;
3290016                 break;
3290017               }
3290018           }
3290019         if (!found)
3290020           {
3290021             break;
3290022           }
3290023       }
3290024     return i;
3290025 }
```

101.11.22 lib/string/strstr.c

Si veda la sezione 94.119.

```
3300001 #include <string.h>
3300002 //----------------------------------------------------------------------
3300003 char *
3300004 strstr (const char *string, const char *substring)
3300005 {
3300006     size_t i;
3300007     size_t j;
3300008     size_t k;
3300009     int found;
3300010     if (substring[0] == 0)
3300011       {
```

```
3300012           return (char *) string;
3300013         }
3300014       for (i = 0, j = 0, found = 0; string[i] != 0; i++)
3300015         {
3300016           if (string[i] == substring[0])
3300017             {
3300018               for (k = i, j = 0;
3300019                    string[k] == substring[j] &&
3300020                    string[k] != 0            &&
3300021                    substring[j] != 0;
3300022                    j++, k++)
3300023                 {
3300024                   ;
3300025                 }
3300026               if (substring[j] == 0)
3300027                 {
3300028                   found = 1;
3300029                 }
3300030             }
3300031           if (found)
3300032             {
3300033               return (char *) (string + i);
3300034             }
3300035         }
3300036       return NULL;
3300037   }
```

101.11.23 lib/string/strtok.c

«

Si veda la sezione 94.120.

```
3310001   #include <string.h>
3310002   //----------------------------------------------------------------------
3310003   char *
3310004   strtok (char *restrict string, const char *restrict delim)
3310005   {
3310006       static char *next = NULL;
3310007       size_t i = 0;
3310008       size_t j;
3310009       int found_token;
3310010       int found_delim;
3310011       //
3310012       // If the string received a the first parameter is a null pointer,
3310013       // the static pointer is used. But if it is already NULL,
3310014       // the scan cannot start.
3310015       //
3310016       if (string == NULL)
3310017         {
```

```
3310018             if (next == NULL)
3310019               {
3310020                 return NULL;
3310021               }
3310022             else
3310023               {
3310024                 string = next;
3310025               }
3310026           }
3310027         //
3310028         // If the string received as the first parameter is empty, the scan
3310029         // cannot start.
3310030         //
3310031         if (string[0] == 0)
3310032           {
3310033             next = NULL;
3310034             return NULL;
3310035           }
3310036         else
3310037           {
3310038             if (delim[0] == 0)
3310039               {
3310040                 return string;
3310041               }
3310042           }
3310043         //
3310044         // Find the next token.
3310045         //
3310046         for (i = 0, found_token = 0, j = 0;
3310047              string[i] != 0 && (!found_token); i++)
3310048           {
3310049             //
3310050             // Look inside delimiters.
3310051             //
3310052             for (j = 0, found_delim = 0; delim[j] != 0; j++)
3310053               {
3310054                 if (string[i] == delim[j])
3310055                   {
3310056                     found_delim = 1;
3310057                   }
3310058               }
3310059             //
3310060             // If current character inside the string is not a delimiter,
3310061             // it is the start of a new token.
3310062             //
3310063             if (!found_delim)
3310064               {
3310065                 found_token = 1;
```

```
3310066                 break;
3310067               }
3310068           }
3310069         //
3310070         // If a token was found, the pointer is updated.
3310071         // If otherwise the token is not found, this means that
3310072         // there are no more.
3310073         //
3310074         if (found_token)
3310075           {
3310076             string += i;
3310077           }
3310078         else
3310079           {
3310080             next = NULL;
3310081             return NULL;
3310082           }
3310083         //
3310084         // Find the end of the token.
3310085         //
3310086         for (i = 0, found_delim = 0; string[i] != 0; i++)
3310087           {
3310088             for (j = 0; delim[j] != 0; j++)
3310089               {
3310090                 if (string[i] == delim[j])
3310091                   {
3310092                     found_delim = 1;
3310093                     break;
3310094                   }
3310095               }
3310096             if (found_delim)
3310097               {
3310098                 break;
3310099               }
3310100           }
3310101         //
3310102         // If a delimiter was found, the corresponding character must be
3310103         // reset to zero. If otherwise the string is terminated, the
3310104         // scan is terminated.
3310105         //
3310106         if (found_delim)
3310107           {
3310108             string[i] = 0;
3310109             next = &string[i+1];
3310110           }
3310111         else
3310112           {
3310113             next = NULL;
```

```
3310114         }
3310115     //
3310116     // At this point, the current string represent the token found.
3310117     //
3310118     return string;
3310119 }
```

101.11.24 lib/string/strxfrm.c

Si veda la sezione 94.123.

```
3320001 #include <string.h>
3320002 //----------------------------------------------------------------------
3320003 size_t
3320004 strxfrm (char *restrict dst, const char *restrict org, size_t n)
3320005 {
3320006     size_t i;
3320007     if (n == 0 && dst == NULL)
3320008       {
3320009         return strlen (org);
3320010       }
3320011     else
3320012       {
3320013         for (i = 0; i < n ; i++)
3320014           {
3320015             dst[i] = org[i];
3320016             if (org[i] == 0)
3320017               {
3320018                 break;
3320019               }
3320020           }
3320021         return i;
3320022       }
3320023 }
```

101.12 os16: «lib/sys/os16.h»

Si veda la sezione 97.2.

```
3330001 #ifndef _SYS_OS16_H
3330002 #define _SYS_OS16_H      1
3330003 //----------------------------------------------------------------------
3330004 // This file contains all the declarations that don't have a better
3330005 // place inside standard headers files. Even declarations related to
3330006 // device numbers and system calls is contained here.
3330007 //----------------------------------------------------------------------
3330008 // Please remember that system calls should never be used (called)
```

```
3330009 // inside the kernel code, because system calls cannot be nested for
3330010 // the os16 simple architecture!
3330011 // If a particular function is necessary inside the kernel, that usually
3330012 // is made by a system call, an appropriate k_...() function must be
3330013 // made, to avoid the problem.
3330014 //----------------------------------------------------------------------
3330015 
3330016 #include <sys/types.h>
3330017 #include <sys/stat.h>
3330018 #include <stdint.h>
3330019 #include <signal.h>
3330020 #include <limits.h>
3330021 #include <stdio.h>
3330022 #include <stdint.h>
3330023 #include <stddef.h>
3330024 #include <const.h>
3330025 #include <restrict.h>
3330026 #include <stdarg.h>
3330027 //----------------------------------------------------------------------
3330028 // Device numbers.
3330029 //----------------------------------------------------------------------
3330030 #define DEV_UNDEFINED_MAJOR     0x00
3330031 #define DEV_UNDEFINED           0x0000
3330032 #define DEV_MEM_MAJOR           0x01
3330033 #define DEV_MEM                 0x0101
3330034 #define DEV_NULL                0x0102
3330035 #define DEV_PORT                0x0103
3330036 #define DEV_ZERO                0x0104
3330037 #define DEV_TTY_MAJOR           0x02
3330038 #define DEV_TTY                 0x0200
3330039 #define DEV_DSK_MAJOR           0x03
3330040 #define DEV_DSK0                0x0300
3330041 #define DEV_DSK1                0x0301
3330042 #define DEV_DSK2                0x0302
3330043 #define DEV_DSK3                0x0303
3330044 #define DEV_KMEM_MAJOR          0x04
3330045 #define DEV_KMEM_PS             0x0401
3330046 #define DEV_KMEM_MMP            0x0402
3330047 #define DEV_KMEM_SB             0x0403
3330048 #define DEV_KMEM_INODE          0x0404
3330049 #define DEV_KMEM_FILE           0x0405
3330050 #define DEV_CONSOLE_MAJOR       0x05
3330051 #define DEV_CONSOLE             0x05FF
3330052 #define DEV_CONSOLE0            0x0500
3330053 #define DEV_CONSOLE1            0x0501
3330054 #define DEV_CONSOLE2            0x0502
3330055 #define DEV_CONSOLE3            0x0503
3330056 #define DEV_CONSOLE4            0x0504
```

```
3330057 //----------------------------------------------------------------------
3330058 // Current segments.
3330059 //----------------------------------------------------------------------
3330060 uint16_t _seg_i (void);
3330061 uint16_t _seg_d (void);
3330062 uint16_t _cs    (void);
3330063 uint16_t _ds    (void);
3330064 uint16_t _ss    (void);
3330065 uint16_t _es    (void);
3330066 uint16_t _sp    (void);
3330067 uint16_t _bp    (void);
3330068 #define seg_i() ((unsigned int) _seg_i ())
3330069 #define seg_d() ((unsigned int) _seg_d ())
3330070 #define cs()    ((unsigned int) _cs ())
3330071 #define ds()    ((unsigned int) _ds ())
3330072 #define ss()    ((unsigned int) _ss ())
3330073 #define es()    ((unsigned int) _es ())
3330074 #define sp()    ((unsigned int) _sp ())
3330075 #define bp()    ((unsigned int) _bp ())
3330076 //----------------------------------------------------------------------
3330077 #define min(a, b) (a < b ? a : b)
3330078 #define max(a, b) (a > b ? a : b)
3330079 //----------------------------------------------------------------------
3330080 #define INPUT_LINE_HIDDEN 0
3330081 #define INPUT_LINE_ECHO   1
3330082 #define INPUT_LINE_STARS  2
3330083 //----------------------------------------------------------------------
3330084 #define MOUNT_DEFAULT     0  // Default mount options.
3330085 #define MOUNT_RO          1  // Read only mount option.
3330086 //----------------------------------------------------------------------
3330087 #define SYS_0                        0  // Nothing to do.
3330088 #define SYS_CHDIR                    1
3330089 #define SYS_CHMOD                    2
3330090 #define SYS_CLOCK                    3
3330091 #define SYS_CLOSE                    4
3330092 #define SYS_EXEC                     5
3330093 #define SYS_EXIT                     6  // [1] see below.
3330094 #define SYS_FCHMOD                   7
3330095 #define SYS_FORK                     8
3330096 #define SYS_FSTAT                    9
3330097 #define SYS_KILL                    10
3330098 #define SYS_LSEEK                   11
3330099 #define SYS_MKDIR                   12
3330100 #define SYS_MKNOD                   13
3330101 #define SYS_MOUNT                   14
3330102 #define SYS_OPEN                    15
3330103 #define SYS_PGRP                    16
3330104 #define SYS_READ                    17
```

```
3330105 #define SYS_SETEUID          18
3330106 #define SYS_SETUID           19
3330107 #define SYS_SIGNAL           20
3330108 #define SYS_SLEEP            21
3330109 #define SYS_STAT             22
3330110 #define SYS_TIME             23
3330111 #define SYS_UAREA            24
3330112 #define SYS_UMASK            25
3330113 #define SYS_UMOUNT           26
3330114 #define SYS_WAIT             27
3330115 #define SYS_WRITE            28
3330116 #define SYS_ZPCHAR           29  // [2] see below.
3330117 #define SYS_ZPSTRING         30  // [2]
3330118 #define SYS_CHOWN            31
3330119 #define SYS_DUP              33
3330120 #define SYS_DUP2             34
3330121 #define SYS_LINK             35
3330122 #define SYS_UNLINK           36
3330123 #define SYS_FCNTL            37
3330124 #define SYS_STIME            38
3330125 #define SYS_FCHOWN           39
3330126 //
3330127 // [1] The files 'crt0...' need to know the value used for the
3330128 //     exit system call. If this value is modified, all the file
3330129 //     'crt0...' have also to be modified the same way.
3330130 //
3330131 // [2] These system calls were developed at the beginning, when no
3330132 //     standard I/O was available. They are to be considered as a
3330133 //     last resort for debugging purposes.
3330134 //
3330135 //----------------------------------------------------------------------
3330136 typedef struct {
3330137     char          path[PATH_MAX];
3330138     int           ret;
3330139     int           errno;
3330140     int           errln;
3330141     char          errfn[PATH_MAX];
3330142 } sysmsg_chdir_t;
3330143 //----------------------------------------------------------------------
3330144 typedef struct {
3330145     char          path[PATH_MAX];
3330146     mode_t        mode;
3330147     int           ret;
3330148     int           errno;
3330149     int           errln;
3330150     char          errfn[PATH_MAX];
3330151 } sysmsg_chmod_t;
3330152 //----------------------------------------------------------------------
```

```
3330153 typedef struct {
3330154     char         path[PATH_MAX];
3330155     uid_t        uid;
3330156     uid_t        gid;
3330157     int          ret;
3330158     int          errno;
3330159     int          errln;
3330160     char         errfn[PATH_MAX];
3330161 } sysmsg_chown_t;
3330162 //----------------------------------------------------------------------
3330163 typedef struct {
3330164     clock_t      ret;
3330165 } sysmsg_clock_t;
3330166 //----------------------------------------------------------------------
3330167 typedef struct {
3330168     int          fdn;
3330169     int          ret;
3330170     int          errno;
3330171     int          errln;
3330172     char         errfn[PATH_MAX];
3330173 } sysmsg_close_t;
3330174 //----------------------------------------------------------------------
3330175 typedef struct {
3330176     int          fdn_old;
3330177     int          ret;
3330178     int          errno;
3330179     int          errln;
3330180     char         errfn[PATH_MAX];
3330181 } sysmsg_dup_t;
3330182 //----------------------------------------------------------------------
3330183 typedef struct {
3330184     int          fdn_old;
3330185     int          fdn_new;
3330186     int          ret;
3330187     int          errno;
3330188     int          errln;
3330189     char         errfn[PATH_MAX];
3330190 } sysmsg_dup2_t;
3330191 //----------------------------------------------------------------------
3330192 typedef struct {
3330193     char         path[PATH_MAX];
3330194     int          argc;
3330195     int          envc;
3330196     char         arg_data[ARG_MAX/2];
3330197     char         env_data[ARG_MAX/2];
3330198     uid_t        uid;
3330199     uid_t        euid;
3330200     int          ret;
```

```
3330201     int          errno;
3330202     int          errln;
3330203     char         errfn[PATH_MAX];
3330204 } sysmsg_exec_t;
3330205 //----------------------------------------------------------------------
3330206 typedef struct {
3330207     int          status;
3330208 } sysmsg_exit_t;
3330209 //----------------------------------------------------------------------
3330210 typedef struct {
3330211     int          fdn;
3330212     mode_t       mode;
3330213     int          ret;
3330214     int          errno;
3330215     int          errln;
3330216     char         errfn[PATH_MAX];
3330217 } sysmsg_fchmod_t;
3330218 //----------------------------------------------------------------------
3330219 typedef struct {
3330220     int          fdn;
3330221     uid_t        uid;
3330222     uid_t        gid;
3330223     int          ret;
3330224     int          errno;
3330225     int          errln;
3330226     char         errfn[PATH_MAX];
3330227 } sysmsg_fchown_t;
3330228 //----------------------------------------------------------------------
3330229 typedef struct {
3330230     int          fdn;
3330231     int          cmd;
3330232     int          arg;
3330233     int          ret;
3330234     int          errno;
3330235     int          errln;
3330236     char         errfn[PATH_MAX];
3330237 } sysmsg_fcntl_t;
3330238 //----------------------------------------------------------------------
3330239 typedef struct {
3330240     pid_t        ret;
3330241     int          errno;
3330242     int          errln;
3330243     char         errfn[PATH_MAX];
3330244 } sysmsg_fork_t;
3330245 //----------------------------------------------------------------------
3330246 typedef struct {
3330247     int          fdn;
3330248     struct stat  stat;
```

```
3330249     int           ret;
3330250     int           errno;
3330251     int           errln;
3330252     char          errfn[PATH_MAX];
3330253 } sysmsg_fstat_t;
3330254 //----------------------------------------------------------------------
3330255 typedef struct {
3330256     pid_t         pid;
3330257     int           signal;
3330258     int           ret;
3330259     int           errno;
3330260     int           errln;
3330261     char          errfn[PATH_MAX];
3330262 } sysmsg_kill_t;
3330263 //----------------------------------------------------------------------
3330264 typedef struct {
3330265     char          path_old[PATH_MAX];
3330266     char          path_new[PATH_MAX];
3330267     int           ret;
3330268     int           errno;
3330269     int           errln;
3330270     char          errfn[PATH_MAX];
3330271 } sysmsg_link_t;
3330272 //----------------------------------------------------------------------
3330273 typedef struct {
3330274     int           fdn;
3330275     off_t         offset;
3330276     int           whence;
3330277     int           ret;
3330278     int           errno;
3330279     int           errln;
3330280     char          errfn[PATH_MAX];
3330281 } sysmsg_lseek_t;
3330282 //----------------------------------------------------------------------
3330283 typedef struct {
3330284     char          path[PATH_MAX];
3330285     mode_t        mode;
3330286     int           ret;
3330287     int           errno;
3330288     int           errln;
3330289     char          errfn[PATH_MAX];
3330290 } sysmsg_mkdir_t;
3330291 //----------------------------------------------------------------------
3330292 typedef struct {
3330293     char          path[PATH_MAX];
3330294     mode_t        mode;
3330295     dev_t         device;
3330296     int           ret;
```

```
3330297     int           errno;
3330298     int           errln;
3330299     char          errfn[PATH_MAX];
3330300 } sysmsg_mknod_t;
3330301 //----------------------------------------------------------------------
3330302 typedef struct {
3330303     char          path_dev[PATH_MAX];
3330304     char          path_mnt[PATH_MAX];
3330305     int           options;
3330306     int           ret;
3330307     int           errno;
3330308     int           errln;
3330309     char          errfn[PATH_MAX];
3330310 } sysmsg_mount_t;
3330311 //----------------------------------------------------------------------
3330312 typedef struct {
3330313     char          path[PATH_MAX];
3330314     int           flags;
3330315     mode_t        mode;
3330316     int           ret;
3330317     int           errno;
3330318     int           errln;
3330319     char          errfn[PATH_MAX];
3330320 } sysmsg_open_t;
3330321 //----------------------------------------------------------------------
3330322 typedef struct {
3330323     int           fdn;
3330324     char          buffer[BUFSIZ];
3330325     size_t        count;
3330326     int           eof;
3330327     ssize_t       ret;
3330328     int           errno;
3330329     int           errln;
3330330     char          errfn[PATH_MAX];
3330331 } sysmsg_read_t;
3330332 //----------------------------------------------------------------------
3330333 typedef struct {
3330334     uid_t         euid;
3330335     int           ret;
3330336     int           errno;
3330337     int           errln;
3330338     char          errfn[PATH_MAX];
3330339 } sysmsg_seteuid_t;
3330340 //----------------------------------------------------------------------
3330341 typedef struct {
3330342     uid_t         uid;
3330343     uid_t         euid;
3330344     uid_t         suid;
```

```
3330345     int          ret;
3330346     int          errno;
3330347     int          errln;
3330348     char         errfn[PATH_MAX];
3330349 } sysmsg_setuid_t;
3330350 //----------------------------------------------------------------------
3330351 typedef struct {
3330352     sighandler_t handler;
3330353     int          signal;
3330354     sighandler_t ret;
3330355     int          errno;
3330356     int          errln;
3330357     char         errfn[PATH_MAX];
3330358 } sysmsg_signal_t;
3330359 //----------------------------------------------------------------------
3330360 #define WAKEUP_EVENT_SIGNAL  1    // 1, 2, 4, 8, 16,...
3330361 #define WAKEUP_EVENT_TIMER   2    // so that can be 'OR' combined.
3330362 #define WAKEUP_EVENT_TTY     4    //
3330363 typedef struct {
3330364     int          events;
3330365     int          signal;
3330366     unsigned int seconds;
3330367     time_t       ret;
3330368 } sysmsg_sleep_t;
3330369 //----------------------------------------------------------------------
3330370 typedef struct {
3330371     char         path[PATH_MAX];
3330372     struct stat  stat;
3330373     int          ret;
3330374     int          errno;
3330375     int          errln;
3330376     char         errfn[PATH_MAX];
3330377 } sysmsg_stat_t;
3330378 //----------------------------------------------------------------------
3330379 typedef struct {
3330380     time_t       ret;
3330381 } sysmsg_time_t;
3330382 //----------------------------------------------------------------------
3330383 typedef struct {
3330384     time_t       timer;
3330385     int          ret;
3330386 } sysmsg_stime_t;
3330387 //----------------------------------------------------------------------
3330388 typedef struct {
3330389     uid_t        uid;          // Read user ID.
3330390     uid_t        euid;         // Effective user ID.
3330391     uid_t        suid;         // Saved user ID.
3330392     pid_t        pid;          // Process ID.
```

```
3330393     pid_t         ppid;         // Parent PID.
3330394     pid_t         pgrp;         // Process group.
3330395     mode_t        umask;        // Access permission mask.
3330396     char          path_cwd[PATH_MAX];
3330397 } sysmsg_uarea_t;
3330398 //----------------------------------------------------------------------
3330399 typedef struct {
3330400     mode_t        umask;
3330401     mode_t        ret;
3330402 } sysmsg_umask_t;
3330403 //----------------------------------------------------------------------
3330404 typedef struct {
3330405     char          path_mnt[PATH_MAX];
3330406     int           ret;
3330407     int           errno;
3330408     int           errln;
3330409     char          errfn[PATH_MAX];
3330410 } sysmsg_umount_t;
3330411 //----------------------------------------------------------------------
3330412 typedef struct {
3330413     char          path[PATH_MAX];
3330414     int           ret;
3330415     int           errno;
3330416     int           errln;
3330417     char          errfn[PATH_MAX];
3330418 } sysmsg_unlink_t;
3330419 //----------------------------------------------------------------------
3330420 typedef struct {
3330421     int           status;
3330422     pid_t         ret;
3330423     int           errno;
3330424     int           errln;
3330425     char          errfn[PATH_MAX];
3330426 } sysmsg_wait_t;
3330427 //----------------------------------------------------------------------
3330428 typedef struct {
3330429     int           fdn;
3330430     char          buffer[BUFSIZ];
3330431     size_t        count;
3330432     ssize_t       ret;
3330433     int           errno;
3330434     int           errln;
3330435     char          errfn[PATH_MAX];
3330436 } sysmsg_write_t;
3330437 //----------------------------------------------------------------------
3330438 typedef struct {
3330439     char          c;
3330440 } sysmsg_zpchar_t;
```

```
3330441 //----------------------------------------------------------------------
3330442 typedef struct {
3330443     char          string[BUFSIZ];
3330444 } sysmsg_zpstring_t;
3330445 //----------------------------------------------------------------------
3330446 void  heap_clear   (void);
3330447 int   heap_min     (void);
3330448 void  input_line   (char *line, char *prompt, size_t size, int type);
3330449 int   mount        (const char *path_dev, const char *path_mnt,
3330450                    int options);
3330451 int   namep        (const char *name, char *path, size_t size);
3330452 void  process_info (void);
3330453 void  sys          (int syscallnr, void *message, size_t size);
3330454 int   umount       (const char *path_mnt);
3330455 void  z_perror     (const char *string);
3330456 int   z_printf     (const char *restrict format, ...);
3330457 int   z_putchar    (int c);
3330458 int   z_puts       (const char *string);
3330459 int   z_vprintf    (const char *restrict format, va_list arg);
3330460 //int  z_vsprintf   (char *restrict string, const char *restrict format,
3330461 //                   va_list arg);
3330462 //----------------------------------------------------------------------
3330463 
3330464 #endif
```

101.12.1 lib/sys/os16/_bp.s

«

Si veda la sezione 94.12.

```
3340001 .global __bp
3340002 .text
3340003 ;----------------------------------------------------------------------
3340004 ; Read the base pointer, as it is before this call.
3340005 ;----------------------------------------------------------------------
3340006 .align 2
3340007 __bp:
3340008     enter #2, #0        ; 1 local variable.
3340009     pushf
3340010     cli
3340011     pusha
3340012     mov   ax, [bp]      ; The previous BP value is saved at *BP.
3340013     mov   -2[bp], ax    ; Save the calculated old SP value.
3340014     popa
3340015     popf
3340016     mov   ax, -2[bp]    ; AX is the function return value.
3340017     leave
3340018     ret
```

101.12.2 lib/sys/os16/_cs.s

«

Si veda la sezione 94.12.

```
3350001 .global __cs
3350002 .text
3350003 ;----------------------------------------------------------------------
3350004 ; Read the code segment value.
3350005 ;----------------------------------------------------------------------
3350006 .align 2
3350007 __cs:
3350008     mov ax, cs
3350009     ret
```

101.12.3 lib/sys/os16/_ds.s

«

Si veda la sezione 94.12.

```
3360001 .global __ds
3360002 .text
3360003 ;----------------------------------------------------------------------
3360004 ; Read the data segment value.
3360005 ;----------------------------------------------------------------------
3360006 .align 2
3360007 __ds:
3360008     mov ax, ds
3360009     ret
```

101.12.4 lib/sys/os16/_es.s

«

Si veda la sezione 94.12.

```
3370001 .global __es
3370002 .text
3370003 ;----------------------------------------------------------------------
3370004 ; Read the extra segment value.
3370005 ;----------------------------------------------------------------------
3370006 .align 2
3370007 __es:
3370008     mov ax, es
3370009     ret
```

101.12.5 lib/sys/os16/_seg_d.s

Si veda la sezione 94.91.

```
3380001 .global __seg_d
3380002 .text
3380003 ;----------------------------------------------------------------------
3380004 ; Read the data segment value.
3380005 ;----------------------------------------------------------------------
3380006 .align 2
3380007 __seg_d:
3380008     mov ax, ds
3380009     ret
```

101.12.6 lib/sys/os16/_seg_i.s

Si veda la sezione 94.91.

```
3390001 .global __seg_i
3390002 .text
3390003 ;----------------------------------------------------------------------
3390004 ; Read the instruction segment value.
3390005 ;----------------------------------------------------------------------
3390006 .align 2
3390007 __seg_i:
3390008     mov ax, cs
3390009     ret
```

101.12.7 lib/sys/os16/_sp.s

Si veda la sezione 94.12.

```
3400001 .global __sp
3400002 .text
3400003 ;----------------------------------------------------------------------
3400004 ; Read the stack pointer, as it is before this call.
3400005 ;----------------------------------------------------------------------
3400006 .align 2
3400007 __sp:
3400008     enter #2, #0        ; 1 local variable.
3400009     pushf
3400010     cli
3400011     pusha
3400012     mov   ax, bp        ; The previous SP is equal to BP + 2 + 2.
3400013     add   ax, #4        ;
3400014     mov   -2[bp], ax    ; Save the calculated old SP value.
3400015     popa
3400016     popf
```

```
3400017     mov    ax, -2[bp]     ; AX is the function return value.
3400018     leave
3400019     ret
```

101.12.8 lib/sys/os16/_ss.s

Si veda la sezione 94.12.

```
3410001 .global __ss
3410002 .text
3410003 ;------------------------------------------------------------------------
3410004 ; Read the stack segment value.
3410005 ;------------------------------------------------------------------------
3410006 .align 2
3410007 __ss:
3410008     mov ax, ss
3410009     ret
```

101.12.9 lib/sys/os16/heap_clear.c

Si veda la sezione 94.57.

```
3420001 #include <sys/os16.h>
3420002 //----------------------------------------------------------------------
3420003 extern uint16_t _end;
3420004 //----------------------------------------------------------------------
3420005 void heap_clear (void)
3420006 {
3420007     uint16_t *a = &_end;
3420008     uint16_t *z = (void *) (sp () - 2);
3420009     for (; a < z; a++)
3420010       {
3420011         *a = 0xFFFF;
3420012       }
3420013 }
```

101.12.10 lib/sys/os16/heap_min.c

Si veda la sezione 94.57.

```
3430001 #include <sys/os16.h>
3430002 //----------------------------------------------------------------------
3430003 extern uint16_t _end;
3430004 //----------------------------------------------------------------------
3430005 int heap_min (void)
3430006 {
```

```
3430007     uint16_t *a = &_end;
3430008     uint16_t *z = (void *) (sp () - 2);
3430009     int      count;
3430010     for (count = 0; a < z && *a == 0xFFFF; a++, count++);
3430011     return (count * 2);
3430012 }
```

101.12.11 lib/sys/os16/input_line.c

Si veda la sezione 94.60.

```
3440001 #include <sys/os16.h>
3440002 #include <string.h>
3440003 #include <stdio.h>
3440004 //----------------------------------------------------------------------
3440005 void
3440006 input_line (char *line, char *prompt, size_t size, int type)
3440007 {
3440008     int    i;     // Index inside the `line[]' array.
3440009     int    c;     // Character received from keyboard.
3440010 
3440011     if (prompt != NULL || strlen (prompt) > 0)
3440012       {
3440013         printf ("%s ", prompt);
3440014       }
3440015     //
3440016     // Loop for character input. Please note that the loop
3440017     // will exit only through `break', where the input line
3440018     // will also be correctly terminated with '\0'.
3440019     //
3440020     for (i = 0;; i++)
3440021       {
3440022         c = getchar ();
3440023         //
3440024         // Control codes.
3440025         //
3440026         if (c == EOF)
3440027           {
3440028             line[i] = 0;
3440029             break;
3440030           }
3440031         else if (c == 4)          // [Ctrl D]
3440032           {
3440033             line[i] = 0;
3440034             break;
3440035           }
3440036         else if (c == 10)         // [Enter]
3440037           {
```

```
3440038                 line[i] = 0;
3440039                 break;
3440040               }
3440041             else if (c == 8)        // [Backspace]
3440042               {
3440043                 if (i == 0)
3440044                   {
3440045                     //
3440046                     // It is already the lowest position, so the video
3440047                     // cursor is moved forward again, so that the prompt
3440048                     // is not overwritten.
3440049                     // The index is set to -1, so that on the next loop,
3440050                     // it will be again zero.
3440051                     //
3440052                     printf (" ");
3440053                     i = -1;
3440054                   }
3440055                 else
3440056                   {
3440057                     i -= 2;
3440058                   }
3440059                 continue;
3440060               }
3440061             //
3440062             // If 'i' is equal 'size - 1', it is not allowed to continue
3440063             // typing.
3440064             //
3440065             if (i == (size - 1))
3440066               {
3440067                 //
3440068                 // Ignore typing, move back the cursor, delete the character
3440069                 // typed and move back again.
3440070                 //
3440071                 printf ("\b \b");
3440072                 i--;
3440073                 continue;
3440074               }
3440075             //
3440076             // Typing is allowed.
3440077             //
3440078             line[i] = c;
3440079             //
3440080             // Verify if it should be hidden.
3440081             //
3440082             if (type == INPUT_LINE_HIDDEN)
3440083               {
3440084                 printf ("\b ");     // Space: at least you see something.
3440085               }
```

```
3440086         else if (type == INPUT_LINE_STARS)
3440087           {
3440088             printf ("\b*");
3440089           }
3440090       }
3440091 }
```

101.12.12 lib/sys/os16/mount.c

«

Si veda la sezione 93.27.

```
3450001 #include <sys/types.h>
3450002 #include <errno.h>
3450003 #include <sys/os16.h>
3450004 #include <stddef.h>
3450005 #include <string.h>
3450006 #include <const.h>
3450007 //----------------------------------------------------------------------
3450008 int
3450009 mount (const char *path_dev, const char *path_mnt, int options)
3450010 {
3450011     sysmsg_mount_t msg;
3450012     //
3450013     strncpy (msg.path_dev, path_dev, PATH_MAX);
3450014     strncpy (msg.path_mnt, path_mnt, PATH_MAX);
3450015     msg.options = options;
3450016     msg.ret     = 0;
3450017     msg.errno   = 0;
3450018     //
3450019     sys (SYS_MOUNT, &msg, (sizeof msg));
3450020     //
3450021     errno = msg.errno;
3450022     errln = msg.errln;
3450023     strncpy (errfn, msg.errfn, PATH_MAX);
3450024     return (msg.ret);
3450025 }
```

101.12.13 lib/sys/os16/namep.c

«

Si veda la sezione 94.74.

```
3460001 #include <sys/os16.h>
3460002 #include <stdlib.h>
3460003 #include <errno.h>
3460004 #include <unistd.h>
3460005 //----------------------------------------------------------------------
3460006 int
```

```
3460007 namep (const char *name, char *path, size_t size)
3460008 {
3460009     char  command[PATH_MAX];
3460010     char *env_path;
3460011     int   p;            // Index used inside the path environment.
3460012     int   c;            // Index used inside the command string.
3460013     int   status;
3460014     //
3460015     // Check for valid input.
3460016     //
3460017     if (name == NULL || name[0] == 0 || path == NULL || name == path)
3460018       {
3460019         errset (EINVAL);                // Invalid argument.
3460020         return (-1);
3460021       }
3460022     //
3460023     // Check if the original command contains at least a '/'. Otherwise
3460024     // a scan for the environment variable 'PATH' must be done.
3460025     //
3460026     if (strchr (name, '/') == NULL)
3460027       {
3460028         //
3460029         // Ok: no '/' there. Get the environment variable 'PATH'.
3460030         //
3460031         env_path = getenv ("PATH");
3460032         if (env_path == NULL)
3460033           {
3460034             //
3460035             // There is no 'PATH' environment value.
3460036             //
3460037             errset (ENOENT);            // No such file or directory.
3460038             return (-1);
3460039           }
3460040         //
3460041         // Scan paths and try to find a file with that name.
3460042         //
3460043         for (p = 0; env_path[p] != 0;)
3460044           {
3460045             for (c = 0;
3460046                  c < (PATH_MAX - strlen(name) - 2) &&
3460047                  env_path[p] != 0                  &&
3460048                  env_path[p] != ':';
3460049                  c++, p++)
3460050               {
3460051                 command[c] = env_path[p];
3460052               }
3460053             //
3460054             // If the loop is ended because the command array does not
```

```
3460055                  // have enough room for the full path, then must return an
3460056                  // error.
3460057                  //
3460058                  if (env_path[p] != ':' && env_path[p] != 0)
3460059                    {
3460060                      errset (ENAMETOOLONG);  // Filename too long.
3460061                      return (-1);
3460062                    }
3460063                  //
3460064                  // The command array has enough space. At index 'c' must
3460065                  // place a zero, to terminate current string.
3460066                  //
3460067                  command[c] = 0;
3460068                  //
3460069                  // Add the rest of the path.
3460070                  //
3460071                  strcat (command, "/");
3460072                  strcat (command, name);
3460073                  //
3460074                  // Verify to have something with that full path name.
3460075                  //
3460076                  status = access (command, F_OK);
3460077                  if (status == 0)
3460078                    {
3460079                      //
3460080                      // Verify to have enough room inside the destination
3460081                      // path.
3460082                      //
3460083                      if (strlen (command) >= size)
3460084                        {
3460085                          //
3460086                          // Sorry: too big. There must be room also for
3460087                          // the string termination null character.
3460088                          //
3460089                          errset (ENAMETOOLONG);      // Filename too long.
3460090                          return (-1);
3460091                        }
3460092                      //
3460093                      // Copy the path and return.
3460094                      //
3460095                      strncpy (path, command, size);
3460096                      return (0);
3460097                    }
3460098                  //
3460099                  // That path was not good: try again. But before returning
3460100                  // to the external loop, must verify if 'p' is to be
3460101                  // incremented, after a ':', because the external loop
3460102                  // does not touch the index 'p',
```

```
3460103                 //
3460104                 if (env_path[p] == ':')
3460105                   {
3460106                     p++;
3460107                   }
3460108               }
3460109             //
3460110             // At this point, there is no match with the paths.
3460111             //
3460112             errset (ENOENT);                // No such file or directory.
3460113             return (-1);
3460114           }
3460115         //
3460116         // At this point, a path was given and the environment variable
3460117         // 'PATH' was not scanned. Just copy the same path. But must verify
3460118         // that the receiving path has enough room for it.
3460119         //
3460120         if (strlen (name) >= size)
3460121           {
3460122             //
3460123             // Sorry: too big.
3460124             //
3460125             errset (ENAMETOOLONG);  // Filename too long.
3460126             return (-1);
3460127           }
3460128         //
3460129         // Ok: copy and return.
3460130         //
3460131         strncpy (path, name, size);
3460132         return (0);
3460133     }
```

101.12.14 lib/sys/os16/process_info.c

«

Si veda la sezione 94.79.

```
3470001     #include <sys/os16.h>
3470002     #include <stdio.h>
3470003
3470004     extern uint16_t _edata;
3470005     extern uint16_t _end;
3470006     //----------------------------------------------------------------------
3470007     void
3470008     process_info (void)
3470009     {
3470010         printf ("cs=%04x ds=%04x ss=%04x es=%04x bp=%04x sp=%04x ",
3470011                 cs (), ds (), ss (), es (), bp (), sp ());
3470012         printf ("edata=%04x ebss=%04x heap=%04x\n",
```

```
3470013                 (int) &_edata, (int) &_end, heap_min ());
3470014 }
```

101.12.15 lib/sys/os16/sys.s

Si veda la sezione 93.37.

```
3480001 .global _sys
3480002 .text
3480003 ;------------------------------------------------------------------------
3480004 ; Call a system call.
3480005 ;
3480006 ; Please remember that system calls should never be used (called) inside
3480007 ; the kernel code, because system calls cannot be nested for the os16
3480008 ; simple architecture!
3480009 ; If a particular function is necessary inside the kernel, that usually
3480010 ; is made by a system call, an appropriate k_...() function must be
3480011 ; made, to avoid the problem.
3480012 ;
3480013 ;------------------------------------------------------------------------
3480014 .align 2
3480015 _sys:
3480016     int   #0x80
3480017     ret
```

101.12.16 lib/sys/os16/umount.c

Si veda la sezione 93.27.

```
3490001 #include <sys/types.h>
3490002 #include <errno.h>
3490003 #include <sys/os16.h>
3490004 #include <stddef.h>
3490005 #include <string.h>
3490006 //----------------------------------------------------------------------
3490007 int
3490008 umount (const char *path_mnt)
3490009 {
3490010     sysmsg_umount_t msg;
3490011     //
3490012     strncpy (msg.path_mnt, path_mnt, PATH_MAX);
3490013     msg.ret      = 0;
3490014     msg.errno    = 0;
3490015     //
3490016     sys (SYS_UMOUNT, &msg, (sizeof msg));
3490017     //
3490018     errno = msg.errno;
```

```
3490019     errln = msg.errln;
3490020     strncpy (errfn, msg.errfn, PATH_MAX);
3490021     return (msg.ret);
3490022 }
```

101.12.17 lib/sys/os16/z_perror.c

«

Si veda la sezione 93.45.

```
3500001 #include <sys/os16.h>
3500002 #include <errno.h>
3500003 #include <stddef.h>
3500004 #include <string.h>
3500005 //----------------------------------------------------------------------
3500006 void
3500007 z_perror (const char *string)
3500008 {
3500009     //
3500010     // If errno is zero, there is nothing to show.
3500011     //
3500012     if (errno == 0)
3500013       {
3500014         return;
3500015       }
3500016     //
3500017     // Show the string if there is one.
3500018     //
3500019     if (string != NULL && strlen (string) > 0)
3500020       {
3500021         z_printf ("%s: ", string);
3500022       }
3500023     //
3500024     // Show the translated error.
3500025     //
3500026     if (errfn[0] != 0 && errln != 0)
3500027       {
3500028         z_printf ("[%s:%u:%i] %s\n",
3500029                   errfn, errln, errno, strerror (errno));
3500030       }
3500031     else
3500032       {
3500033         z_printf ("[%i] %s\n", errno, strerror (errno));
3500034       }
3500035 }
```

101.12.18 lib/sys/os16/z_printf.c

Si veda la sezione 93.45.

```
3510001 #include <sys/os16.h>
3510002 //----------------------------------------------------------------------
3510003 int
3510004 z_printf (char *format, ...)
3510005 {
3510006     va_list ap;
3510007     va_start (ap, format);
3510008     return z_vprintf (format, ap);
3510009 }
```

101.12.19 lib/sys/os16/z_putchar.c

Si veda la sezione 93.45.

```
3520001 #include <sys/os16.h>
3520002 //----------------------------------------------------------------------
3520003 int
3520004 z_putchar (int c)
3520005 {
3520006     sysmsg_zpchar_t msg;
3520007     msg.c = c;
3520008     sys (SYS_ZPCHAR, &msg, (sizeof msg));
3520009     return (c);
3520010 }
```

101.12.20 lib/sys/os16/z_puts.c

Si veda la sezione 93.45.

```
3530001 #include <sys/os16.h>
3530002 //----------------------------------------------------------------------
3530003 int
3530004 z_puts (char *string)
3530005 {
3530006     unsigned int i;
3530007     for (i = 0; string[i] != 0; string++)
3530008       {
3530009         z_putchar ((int) string[i]);
3530010       }
3530011     z_putchar ((int) '\n');
3530012     return (1);
3530013 }
```

101.12.21 lib/sys/os16/z_vprintf.c

Si veda la sezione 93.45.

```
3540001 #include <sys/os16.h>
3540002 //----------------------------------------------------------------------
3540003 int
3540004 z_vprintf (char *format, va_list arg)
3540005 {
3540006     int                        ret;
3540007     sysmsg_zpstring_t msg;
3540008     msg.string[0] = 0;
3540009     ret = vsprintf (msg.string, format, arg);
3540010     sys (SYS_ZPSTRING, &msg, (sizeof msg));
3540011     return ret;
3540012 }
```

101.13 os16: «lib/sys/stat.h»

Si veda la sezione 97.2.

```
3550001 #ifndef _SYS_STAT_H
3550002 #define _SYS_STAT_H     1
3550003
3550004 #include <restrict.h>
3550005 #include <const.h>
3550006 #include <sys/types.h>  // dev_t
3550007                         // off_t
3550008                         // blkcnt_t
3550009                         // blksize_t
3550010                         // ino_t
3550011                         // mode_t
3550012                         // nlink_t
3550013                         // uid_t
3550014                         // gid_t
3550015                         // time_t
3550016 //----------------------------------------------------------------------
3550017 // File type.
3550018 //----------------------------------------------------------------------
3550019 #define S_IFMT   0170000        // File type mask.
3550020 //
3550021 #define S_IFBLK  0060000        // Block device file.
3550022 #define S_IFCHR  0020000        // Character device file.
3550023 #define S_IFIFO  0010000        // Pipe (FIFO) file.
3550024 #define S_IFREG  0100000        // Regular file.
3550025 #define S_IFDIR  0040000        // Directory.
3550026 #define S_IFLNK  0120000        // Symbolic link.
3550027 #define S_IFSOCK 0140000        // Unix domain socket.
```

```
3550028 //----------------------------------------------------------------------
3550029 // Owner user access permissions.
3550030 //----------------------------------------------------------------------
3550031 #define S_IRWXU  0000700        // Owner user access permissions mask.
3550032 //
3550033 #define S_IRUSR  0000400        // Owner user read access permission.
3550034 #define S_IWUSR  0000200        // Owner user write access permission.
3550035 #define S_IXUSR  0000100        // Owner user execution or cross perm.
3550036 //----------------------------------------------------------------------
3550037 // Group owner access permissions.
3550038 //----------------------------------------------------------------------
3550039 #define S_IRWXG  0000070        // Owner group access permissions mask.
3550040 //
3550041 #define S_IRGRP  0000040        // Owner group read access permission.
3550042 #define S_IWGRP  0000020        // Owner group write access permission.
3550043 #define S_IXGRP  0000010        // Owner group execution or cross perm.
3550044 //----------------------------------------------------------------------
3550045 // Other users access permissions.
3550046 //----------------------------------------------------------------------
3550047 #define S_IRWXO  0000007        // Other users access permissions mask.
3550048 //
3550049 #define S_IROTH  0000004        // Other users read access permission.
3550050 #define S_IWOTH  0000002        // Other users write access permissions.
3550051 #define S_IXOTH  0000001        // Other users execution or cross perm.
3550052 //----------------------------------------------------------------------
3550053 // S-bit: in this case there is no mask to select all of them.
3550054 //----------------------------------------------------------------------
3550055 #define S_ISUID  0004000        // S-UID.
3550056 #define S_ISGID  0002000        // S-GID.
3550057 #define S_ISVTX  0001000        // Sticky.
3550058 //----------------------------------------------------------------------
3550059 // Macro-instructions to verify the type of file.
3550060 //----------------------------------------------------------------------
3550061 #define S_ISBLK(m)   (((m) & S_IFMT) == S_IFBLK)   // Block device.
3550062 #define S_ISCHR(m)   (((m) & S_IFMT) == S_IFCHR)   // Character device.
3550063 #define S_ISFIFO(m)  (((m) & S_IFMT) == S_IFIFO)   // FIFO.
3550064 #define S_ISREG(m)   (((m) & S_IFMT) == S_IFREG)   // Regular file.
3550065 #define S_ISDIR(m)   (((m) & S_IFMT) == S_IFDIR)   // Directory.
3550066 #define S_ISLNK(m)   (((m) & S_IFMT) == S_IFLNK)   // Symbolic link.
3550067 #define S_ISSOCK(m)  (((m) & S_IFMT) == S_IFSOCK)  // Socket.
3550068 //----------------------------------------------------------------------
3550069 // Structure 'stat'.
3550070 //----------------------------------------------------------------------
3550071 struct stat {
3550072     dev_t     st_dev;       // Device containing the file.
3550073     ino_t     st_ino;       // File serial number (inode number).
3550074     mode_t    st_mode;      // File type and permissions.
3550075     nlink_t   st_nlink;     // Links to the file.
```

```
3550076     uid_t     st_uid;       // Owner user id.
3550077     gid_t     st_gid;       // Owner group id.
3550078     dev_t     st_rdev;      // Device number if it is a device file.
3550079     off_t     st_size;      // File size.
3550080     time_t    st_atime;     // Last access time.
3550081     time_t    st_mtime;     // Last modification time.
3550082     time_t    st_ctime;     // Last inode modification.
3550083     blksize_t st_blksize;   // Block size for I/O operations.
3550084     blkcnt_t  st_blocks;    // File size / block size.
3550085 };
3550086 //----------------------------------------------------------------------
3550087 // Function prototypes.
3550088 //----------------------------------------------------------------------
3550089 int    chmod  (const char *path, mode_t mode);
3550090 int    fchmod (int fdn, mode_t mode);
3550091 int    fstat  (int fdn, struct stat *buffer);
3550092 int    lstat  (const char *restrict path, struct stat *restrict buffer);
3550093 int    mkdir  (const char *path, mode_t mode);
3550094 int    mkfifo (const char *path, mode_t mode);
3550095 int    mknod  (const char *path, mode_t mode, dev_t dev);
3550096 int    stat   (const char *restrict path, struct stat *restrict buffer);
3550097 mode_t umask  (mode_t mask);
3550098
3550099 #endif // _SYS_STAT_H
```

101.13.1 lib/sys/stat/chmod.c

«

Si veda la sezione 93.4.

```
3560001 #include <sys/stat.h>
3560002 #include <string.h>
3560003 #include <const.h>
3560004 #include <sys/os16.h>
3560005 #include <errno.h>
3560006 #include <limits.h>
3560007 //----------------------------------------------------------------------
3560008 int
3560009 chmod (const char *path, mode_t mode)
3560010 {
3560011     sysmsg_chmod_t msg;
3560012     //
3560013     strncpy (msg.path, path, PATH_MAX);
3560014     msg.mode = mode;
3560015     //
3560016     sys (SYS_CHMOD, &msg, (sizeof msg));
3560017     //
3560018     errno = msg.errno;
3560019     errln = msg.errln;
```

```
3560020     strncpy (errfn, msg.errfn, PATH_MAX);
3560021     return (msg.ret);
3560022 }
```

101.13.2 lib/sys/stat/fchmod.c

Si veda la sezione 93.4.

```
3570001 #include <sys/stat.h>
3570002 #include <string.h>
3570003 #include <const.h>
3570004 #include <sys/os16.h>
3570005 #include <errno.h>
3570006 #include <limits.h>
3570007 //----------------------------------------------------------------------
3570008 int
3570009 fchmod (int fdn, mode_t mode)
3570010 {
3570011     sysmsg_fchmod_t msg;
3570012     //
3570013     msg.fdn  = fdn;
3570014     msg.mode = mode;
3570015     //
3570016     sys (SYS_FCHMOD, &msg, (sizeof msg));
3570017     //
3570018     errno = msg.errno;
3570019     errln = msg.errln;
3570020     strncpy (errfn, msg.errfn, PATH_MAX);
3570021     return (msg.ret);
3570022 }
```

101.13.3 lib/sys/stat/fstat.c

Si veda la sezione 93.36.

```
3580001 #include <unistd.h>
3580002 #include <errno.h>
3580003 #include <sys/os16.h>
3580004 #include <string.h>
3580005 //----------------------------------------------------------------------
3580006 int
3580007 fstat (int fdn, struct stat *buffer)
3580008 {
3580009     sysmsg_fstat_t msg;
3580010     //
3580011     msg.fdn              = fdn;
3580012     msg.stat.st_dev      = buffer->st_dev;
```

```
3580013     msg.stat.st_ino      = buffer->st_ino;
3580014     msg.stat.st_mode     = buffer->st_mode;
3580015     msg.stat.st_nlink    = buffer->st_nlink;
3580016     msg.stat.st_uid      = buffer->st_uid;
3580017     msg.stat.st_gid      = buffer->st_gid;
3580018     msg.stat.st_rdev     = buffer->st_rdev;
3580019     msg.stat.st_size     = buffer->st_size;
3580020     msg.stat.st_atime    = buffer->st_atime;
3580021     msg.stat.st_mtime    = buffer->st_mtime;
3580022     msg.stat.st_ctime    = buffer->st_ctime;
3580023     msg.stat.st_blksize = buffer->st_blksize;
3580024     msg.stat.st_blocks   = buffer->st_blocks;
3580025     //
3580026     sys (SYS_FSTAT, &msg, (sizeof msg));
3580027     //
3580028     buffer->st_dev       = msg.stat.st_dev;
3580029     buffer->st_ino       = msg.stat.st_ino;
3580030     buffer->st_mode      = msg.stat.st_mode;
3580031     buffer->st_nlink     = msg.stat.st_nlink;
3580032     buffer->st_uid       = msg.stat.st_uid;
3580033     buffer->st_gid       = msg.stat.st_gid;
3580034     buffer->st_rdev      = msg.stat.st_rdev;
3580035     buffer->st_size      = msg.stat.st_size;
3580036     buffer->st_atime     = msg.stat.st_atime;
3580037     buffer->st_mtime     = msg.stat.st_mtime;
3580038     buffer->st_ctime     = msg.stat.st_ctime;
3580039     buffer->st_blksize   = msg.stat.st_blksize;
3580040     buffer->st_blocks    = msg.stat.st_blocks;
3580041     //
3580042     errno = msg.errno;
3580043     errln = msg.errln;
3580044     strncpy (errfn, msg.errfn, PATH_MAX);
3580045     return (msg.ret);
3580046 }
```

101.13.4 lib/sys/stat/mkdir.c

Si veda la sezione 93.25.

```
3590001 #include <sys/stat.h>
3590002 #include <string.h>
3590003 #include <const.h>
3590004 #include <sys/os16.h>
3590005 #include <errno.h>
3590006 #include <limits.h>
3590007 //----------------------------------------------------------------------
3590008 int
3590009 mkdir (const char *path, mode_t mode)
```

```
3590010 {
3590011     sysmsg_mkdir_t msg;
3590012     //
3590013     strncpy (msg.path, path, PATH_MAX);
3590014     msg.mode = mode;
3590015     //
3590016     sys (SYS_MKDIR, &msg, (sizeof msg));
3590017     //
3590018     errno = msg.errno;
3590019     errln = msg.errln;
3590020     strncpy (errfn, msg.errfn, PATH_MAX);
3590021     return (msg.ret);
3590022 }
```

101.13.5 lib/sys/stat/mknod.c

Si veda la sezione 93.26.

```
3600001 #include <unistd.h>
3600002 #include <errno.h>
3600003 #include <sys/os16.h>
3600004 #include <string.h>
3600005 //----------------------------------------------------------------------
3600006 int
3600007 mknod (const char *path, mode_t mode, dev_t device)
3600008 {
3600009     sysmsg_mknod_t msg;
3600010     //
3600011     strncpy (msg.path, path, PATH_MAX);
3600012     msg.mode   = mode;
3600013     msg.device = device;
3600014     //
3600015     sys (SYS_MKNOD, &msg, (sizeof msg));
3600016     //
3600017     errno = msg.errno;
3600018     errln = msg.errln;
3600019     strncpy (errfn, msg.errfn, PATH_MAX);
3600020     return (msg.ret);
3600021 }
```

101.13.6 lib/sys/stat/stat.c

Si veda la sezione 93.36.

```
3610001 #include <unistd.h>
3610002 #include <errno.h>
3610003 #include <sys/os16.h>
```

```
3610004 #include <string.h>
3610005 //----------------------------------------------------------------------
3610006 int
3610007 stat (const char *path, struct stat *buffer)
3610008 {
3610009     sysmsg_stat_t msg;
3610010     //
3610011     strncpy (msg.path, path, PATH_MAX);
3610012     //
3610013     msg.stat.st_dev     = buffer->st_dev;
3610014     msg.stat.st_ino     = buffer->st_ino;
3610015     msg.stat.st_mode    = buffer->st_mode;
3610016     msg.stat.st_nlink   = buffer->st_nlink;
3610017     msg.stat.st_uid     = buffer->st_uid;
3610018     msg.stat.st_gid     = buffer->st_gid;
3610019     msg.stat.st_rdev    = buffer->st_rdev;
3610020     msg.stat.st_size    = buffer->st_size;
3610021     msg.stat.st_atime   = buffer->st_atime;
3610022     msg.stat.st_mtime   = buffer->st_mtime;
3610023     msg.stat.st_ctime   = buffer->st_ctime;
3610024     msg.stat.st_blksize = buffer->st_blksize;
3610025     msg.stat.st_blocks  = buffer->st_blocks;
3610026     //
3610027     sys (SYS_STAT, &msg, (sizeof msg));
3610028     //
3610029     buffer->st_dev      = msg.stat.st_dev;
3610030     buffer->st_ino      = msg.stat.st_ino;
3610031     buffer->st_mode     = msg.stat.st_mode;
3610032     buffer->st_nlink    = msg.stat.st_nlink;
3610033     buffer->st_uid      = msg.stat.st_uid;
3610034     buffer->st_gid      = msg.stat.st_gid;
3610035     buffer->st_rdev     = msg.stat.st_rdev;
3610036     buffer->st_size     = msg.stat.st_size;
3610037     buffer->st_atime    = msg.stat.st_atime;
3610038     buffer->st_mtime    = msg.stat.st_mtime;
3610039     buffer->st_ctime    = msg.stat.st_ctime;
3610040     buffer->st_blksize  = msg.stat.st_blksize;
3610041     buffer->st_blocks   = msg.stat.st_blocks;
3610042     //
3610043     errno = msg.errno;
3610044     errln = msg.errln;
3610045     strncpy (errfn, msg.errfn, PATH_MAX);
3610046     return (msg.ret);
3610047 }
```

101.13.7 lib/sys/stat/umask.c

Si veda la sezione 93.40.

```
3620001 #include <sys/stat.h>
3620002 #include <string.h>
3620003 #include <const.h>
3620004 #include <sys/os16.h>
3620005 #include <errno.h>
3620006 #include <limits.h>
3620007 //----------------------------------------------------------------------
3620008 mode_t
3620009 umask (mode_t mask)
3620010 {
3620011     sysmsg_umask_t msg;
3620012     msg.umask = mask;
3620013     sys (SYS_UMASK, &msg, (sizeof msg));
3620014     return (msg.ret);
3620015 }
```

101.14 os16: «lib/sys/types.h»

Si veda la sezione 97.2.

```
3630001 #ifndef _SYS_TYPES_H
3630002 #define _SYS_TYPES_H    1
3630003 //----------------------------------------------------------------------
3630004 
3630005 #include <clock_t.h>
3630006 #include <time_t.h>
3630007 #include <size_t.h>
3630008 #include <stdint.h>
3630009 //----------------------------------------------------------------------
3630010 typedef     long int blkcnt_t;
3630011 typedef     long int blksize_t;
3630012 typedef     uint16_t dev_t;     // Traditional device size.
3630013 typedef unsigned int id_t;
3630014 typedef unsigned int gid_t;
3630015 typedef unsigned int uid_t;
3630016 typedef     uint16_t ino_t;     // Minix 1 file system inode size.
3630017 typedef     uint16_t mode_t;    // Minix 1 file system mode size.
3630018 typedef unsigned int nlink_t;
3630019 typedef     long int off_t;
3630020 typedef          int pid_t;
3630021 typedef unsigned int pthread_t;
3630022 typedef     long int ssize_t;
3630023 //----------------------------------------------------------------------
3630024 // Common extentions.
```

```
3630025 //
3630026 dev_t makedev (int major, int minor);
3630027 int   major   (dev_t device);
3630028 int   minor   (dev_t device);
3630029 //----------------------------------------------------------------------
3630030
3630031 #endif
```

101.14.1 lib/sys/types/major.c

Si veda la sezione 94.65.

```
3640001 #include <sys/types.h>
3640002 //----------------------------------------------------------------------
3640003 int
3640004 major (dev_t device)
3640005 {
3640006     return ((int) (device / 256));
3640007 }
```

101.14.2 lib/sys/types/makedev.c

Si veda la sezione 94.65.

```
3650001 #include <sys/types.h>
3650002 //----------------------------------------------------------------------
3650003 dev_t
3650004 makedev (int major, int minor)
3650005 {
3650006     return ((dev_t) (major * 256 + minor));
3650007 }
```

101.14.3 lib/sys/types/minor.c

Si veda la sezione 94.65.

```
3660001 #include <sys/types.h>
3660002 //----------------------------------------------------------------------
3660003 int
3660004 minor (dev_t device)
3660005 {
3660006     return ((dev_t) (device & 0x00FF));
3660007 }
```

101.15 os16: «lib/sys/wait.h»

Si veda la sezione 97.2.

```
3670001 #ifndef _SYS_WAIT_H
3670002 #define _SYS_WAIT_H      1
3670003
3670004 #include <sys/types.h>
3670005
3670006 //----------------------------------------------------------------------
3670007 pid_t wait    (int *status);
3670008 //----------------------------------------------------------------------
3670009
3670010 #endif
```

101.15.1 lib/sys/wait/wait.c

Si veda la sezione 93.43.

```
3680001 #include <sys/types.h>
3680002 #include <errno.h>
3680003 #include <sys/os16.h>
3680004 #include <stddef.h>
3680005 #include <string.h>
3680006 //----------------------------------------------------------------------
3680007 pid_t
3680008 wait (int *status)
3680009 {
3680010     sysmsg_wait_t msg;
3680011     msg.ret    = 0;
3680012     msg.errno  = 0;
3680013     msg.status = 0;
3680014     while (msg.ret == 0)
3680015       {
3680016         //
3680017         // Loop as long as there are children, an none is dead.
3680018         //
3680019         sys (SYS_WAIT, &msg, (sizeof msg));
3680020       }
3680021     errno = msg.errno;
3680022     errln = msg.errln;
3680023     strncpy (errfn, msg.errfn, PATH_MAX);
3680024     //
3680025     if (status != NULL)
3680026       {
3680027         //
3680028         // Only the low eight bits are returned.
3680029         //
3680030         *status = (msg.status & 0x00FF);
```

```
3680031         }
3680032     return (msg.ret);
3680033 }
```

101.16 os16: «lib/time.h»

«

Si veda la sezione 97.2.

```
3690001 #ifndef _TIME_H
3690002 #define _TIME_H       1
3690003 //----------------------------------------------------------------------
3690004 
3690005 #include <const.h>
3690006 #include <restrict.h>
3690007 #include <size_t.h>
3690008 #include <time_t.h>
3690009 #include <clock_t.h>
3690010 #include <NULL.h>
3690011 #include <stdint.h>
3690012 //----------------------------------------------------------------------
3690013 #define CLOCKS_PER_SEC  18  // Should be 18.22 Hz, but it is a 'int'.
3690014 //----------------------------------------------------------------------
3690015 struct tm {int tm_sec;  int tm_min;  int tm_hour;
3690016            int tm_mday; int tm_mon;  int tm_year;
3690017            int tm_wday; int tm_yday; int tm_isdst;};
3690018 //----------------------------------------------------------------------
3690019 clock_t    clock      (void);
3690020 time_t     time       (time_t *timer);
3690021 int        stime      (time_t *timer);
3690022 double     difftime   (time_t time1, time_t time0);
3690023 time_t     mktime     (struct tm *timeptr);
3690024 struct tm *gmtime     (const time_t *timer);
3690025 struct tm *localtime  (const time_t *timer);
3690026 char      *asctime    (const struct tm *timeptr);
3690027 char      *ctime      (const time_t *timer);
3690028 size_t     strftime   (char * restrict s, size_t maxsize,
3690029                        const char * restrict format,
3690030                        const struct tm * restrict timeptr);
3690031 //----------------------------------------------------------------------
3690032 #define difftime(t1,t0)  ((double)((t1)-(t0)))
3690033 #define ctime(t)         (asctime (localtime (t)))
3690034 #define localtime(t)     (gmtime (t))
3690035 //----------------------------------------------------------------------
3690036 
3690037 #endif
```

101.16.1 lib/time/asctime.c

«

Si veda la sezione 94.13.

```
3700001 #include <time.h>
3700002 #include <string.h>
3700003 #include <stdio.h>
3700004
3700005 //----------------------------------------------------------------------
3700006 char *
3700007 asctime (const struct tm *timeptr)
3700008 {
3700009     static char time_string[25]; // 'Sun Jan 30 24:00:00 2111'
3700010     //
3700011     // Check argument.
3700012     //
3700013     if (timeptr == NULL)
3700014       {
3700015         return (NULL);
3700016       }
3700017     //
3700018     // Set week day.
3700019     //
3700020     switch (timeptr->tm_wday)
3700021       {
3700022         case 0:
3700023             strcpy (&time_string[0], "Sun");
3700024             break;
3700025         case 1:
3700026             strcpy (&time_string[0], "Mon");
3700027             break;
3700028         case 2:
3700029             strcpy (&time_string[0], "Tue");
3700030             break;
3700031         case 3:
3700032             strcpy (&time_string[0], "Wed");
3700033             break;
3700034         case 4:
3700035             strcpy (&time_string[0], "Thu");
3700036             break;
3700037         case 5:
3700038             strcpy (&time_string[0], "Fri");
3700039             break;
3700040         case 6:
3700041             strcpy (&time_string[0], "Sat");
3700042             break;
3700043         default:
3700044             strcpy (&time_string[0], "Err");
3700045       }
```

```
3700046     //
3700047     // Set month.
3700048     //
3700049     switch (timeptr->tm_mon)
3700050       {
3700051         case 1:
3700052             strcpy (&time_string[3], " Jan");
3700053             break;
3700054         case 2:
3700055             strcpy (&time_string[3], " Feb");
3700056             break;
3700057         case 3:
3700058             strcpy (&time_string[3], " Mar");
3700059             break;
3700060         case 4:
3700061             strcpy (&time_string[3], " Apr");
3700062             break;
3700063         case 5:
3700064             strcpy (&time_string[3], " May");
3700065             break;
3700066         case 6:
3700067             strcpy (&time_string[3], " Jun");
3700068             break;
3700069         case 7:
3700070             strcpy (&time_string[3], " Jul");
3700071             break;
3700072         case 8:
3700073             strcpy (&time_string[3], " Aug");
3700074             break;
3700075         case 9:
3700076             strcpy (&time_string[3], " Sep");
3700077             break;
3700078         case 10:
3700079             strcpy (&time_string[3], " Oct");
3700080             break;
3700081         case 11:
3700082             strcpy (&time_string[3], " Nov");
3700083             break;
3700084         case 12:
3700085             strcpy (&time_string[3], " Dec");
3700086             break;
3700087         default:
3700088             strcpy (&time_string[3], " Err");
3700089       }
3700090     //
3700091     // Set day of month, hour, minute, second and year.
3700092     //
3700093     sprintf (&time_string[7], " %2i %2i:%2i:%2i %4i",
```

```
3700094                 timeptr->tm_mday,  timeptr->tm_hour,  timeptr->tm_min,
3700095                 timeptr->tm_sec,  timeptr->tm_year);
3700096     //
3700097     //
3700098     //
3700099     return (&time_string[0]);
3700100 }
```

101.16.2 lib/time/clock.c

Si veda la sezione 93.6.

```
3710001 #include <time.h>
3710002 #include <sys/os16.h>
3710003 //----------------------------------------------------------------------
3710004 clock_t
3710005 clock (void)
3710006 {
3710007     sysmsg_clock_t msg;
3710008     msg.ret = 0;
3710009     sys (SYS_CLOCK, &msg, (sizeof msg));
3710010     return (msg.ret);
3710011 }
3710012
```

101.16.3 lib/time/gmtime.c

Si veda la sezione 94.13.

```
3720001 #include <time.h>
3720002 //----------------------------------------------------------------------
3720003 static int leap_year (int year);
3720004 //----------------------------------------------------------------------
3720005 struct tm *
3720006 gmtime (const time_t *timer)
3720007 {
3720008     static struct tm  tms;
3720009     int               loop;
3720010     unsigned int      remainder;
3720011     unsigned int      days;
3720012     //
3720013     // Check argument.
3720014     //
3720015     if (timer == NULL)
3720016       {
3720017         return (NULL);
3720018       }
```

```
3720019       //
3720020       // Days since epoch. There are 86400 seconds per day.
3720021       // At the moment, the field 'tm_yday' will contain
3720022       // all days since epoch.
3720023       //
3720024       days      = *timer / 86400L;
3720025       remainder = *timer % 86400L;
3720026       //
3720027       // Minutes, after full days.
3720028       //
3720029       tms.tm_min = remainder / 60U;
3720030       //
3720031       // Seconds, after full minutes.
3720032       //
3720033       tms.tm_sec = remainder % 60U;
3720034       //
3720035       // Hours, after full days.
3720036       //
3720037       tms.tm_hour = tms.tm_min / 60;
3720038       //
3720039       // Minutes, after full hours.
3720040       //
3720041       tms.tm_min = tms.tm_min % 60;
3720042       //
3720043       // Find the week day. Must remove some days to align the
3720044       // calculation. So: the week days of the first week of 1970
3720045       // are not valid! After 1970-01-04 calculations are right.
3720046       //
3720047       tms.tm_wday = (days - 3) % 7;
3720048       //
3720049       // Find the year: the field 'tm_yday' will be reduced to the days
3720050       // of current year.
3720051       //
3720052       for (tms.tm_year = 1970; days > 0; tms.tm_year++)
3720053         {
3720054           if (leap_year (tms.tm_year))
3720055             {
3720056               if (days >= 366)
3720057                 {
3720058                   days -= 366;
3720059                   continue;
3720060                 }
3720061               else
3720062                 {
3720063                   break;
3720064                 }
3720065             }
3720066           else
```

```
3720067              {
3720068                if (days >= 365)
3720069                  {
3720070                    days -= 365;
3720071                    continue;
3720072                  }
3720073                else
3720074                  {
3720075                    break;
3720076                  }
3720077              }
3720078          }
3720079        //
3720080        // Day of the year.
3720081        //
3720082        tms.tm_yday = days + 1;
3720083        //
3720084        // Find the month.
3720085        //
3720086        tms.tm_mday = days + 1;
3720087        //
3720088        for (tms.tm_mon = 0, loop = 1; tms.tm_mon <= 12 && loop;)
3720089          {
3720090            tms.tm_mon++;
3720091            //
3720092            switch (tms.tm_mon)
3720093              {
3720094                case 1:
3720095                case 3:
3720096                case 5:
3720097                case 7:
3720098                case 8:
3720099                case 10:
3720100                case 12:
3720101                    if (tms.tm_mday >= 31)
3720102                      {
3720103                        tms.tm_mday -= 31;
3720104                      }
3720105                    else
3720106                      {
3720107                        loop = 0;
3720108                      }
3720109                    break;
3720110                case 4:
3720111                case 6:
3720112                case 9:
3720113                case 11:
3720114                    if (tms.tm_mday >= 30)
```

```
3720115                     {
3720116                       tms.tm_mday -= 30;
3720117                     }
3720118                   else
3720119                     {
3720120                       loop = 0;
3720121                     }
3720122                   break;
3720123               case 2:
3720124                   if (leap_year (tms.tm_year))
3720125                     {
3720126                       if (tms.tm_mday >= 29)
3720127                         {
3720128                           tms.tm_mday -= 29;
3720129                         }
3720130                       else
3720131                         {
3720132                           loop = 0;
3720133                         }
3720134                     }
3720135                   else
3720136                     {
3720137                       if (tms.tm_mday >= 28)
3720138                         {
3720139                           tms.tm_mday -= 28;
3720140                         }
3720141                       else
3720142                         {
3720143                           loop = 0;
3720144                         }
3720145                     }
3720146                   break;
3720147             }
3720148         }
3720149     //
3720150     // No check for day light saving time.
3720151     //
3720152     tms.tm_isdst = 0;
3720153     //
3720154     // Return.
3720155     //
3720156     return (&tms);
3720157 }
3720158 //----------------------------------------------------------------------
3720159 static int
3720160 leap_year (int year)
3720161 {
3720162     if ((year % 4) == 0)
```

```
3720163         {
3720164           if ((year % 100) == 0)
3720165             {
3720166               if ((year % 400) == 0)
3720167                 {
3720168                   return (1);
3720169                 }
3720170               else
3720171                 {
3720172                   return (0);
3720173                 }
3720174             }
3720175           else
3720176             {
3720177               return (1);
3720178             }
3720179         }
3720180       else
3720181         {
3720182           return (0);
3720183         }
3720184 }
```

101.16.4 lib/time/mktime.c

Si veda la sezione 94.13.

```
3730001 #include <time.h>
3730002 #include <string.h>
3730003 #include <stdio.h>
3730004 //----------------------------------------------------------------------
3730005 static int leap_year (int year);
3730006 //----------------------------------------------------------------------
3730007 time_t
3730008 mktime (const struct tm *timeptr)
3730009 {
3730010     time_t timer_total;
3730011     time_t timer_aux;
3730012     int    days;
3730013     int    month;
3730014     int    year;
3730015     //
3730016     // From seconds to days.
3730017     //
3730018     timer_total  = timeptr->tm_sec;
3730019     //
3730020     timer_aux    = timeptr->tm_min;
3730021     timer_aux   *= 60;
```

```
3730022         timer_total += timer_aux;
3730023         //
3730024         timer_aux    = timeptr->tm_hour;
3730025         timer_aux   *= (60 * 60);
3730026         timer_total += timer_aux;
3730027         //
3730028         timer_aux    = timeptr->tm_mday;
3730029         timer_aux   *= 24;
3730030         timer_aux   *= (60 * 60);
3730031         timer_total += timer_aux;
3730032         //
3730033         // Month: add the days of months.
3730034         // Will scan the months, from the first, but before the
3730035         // months of the value inside field `tm_mon'.
3730036         //
3730037         for (month = 1, days = 0; month < timeptr->tm_mon; month++)
3730038           {
3730039             switch (month)
3730040               {
3730041                 case 1:
3730042                 case 3:
3730043                 case 5:
3730044                 case 7:
3730045                 case 8:
3730046                 case 10:
3730047                     //
3730048                     // There is no December, because the scan can go up to
3730049                     // the month before the value inside field `tm_mon'.
3730050                     //
3730051                     days += 31;
3730052                     break;
3730053                 case 4:
3730054                 case 6:
3730055                 case 9:
3730056                 case 11:
3730057                     days += 30;
3730058                     break;
3730059                 case 2:
3730060                     if (leap_year (timeptr->tm_year))
3730061                       {
3730062                         days += 29;
3730063                       }
3730064                     else
3730065                       {
3730066                         days += 28;
3730067                       }
3730068                     break;
3730069               }
```

```
3730070         }
3730071       //
3730072       timer_aux    = days;
3730073       timer_aux   *= 24;
3730074       timer_aux   *= (60 * 60);
3730075       timer_total += timer_aux;
3730076       //
3730077       // Year. The work is similar to the one of months: days of
3730078       // years are counted, up to the year before the one reported
3730079       // by the field 'tm_year'.
3730080       //
3730081       for (year = 1970, days = 0; year < timeptr->tm_year; year++)
3730082         {
3730083           if (leap_year (year))
3730084             {
3730085               days += 366;
3730086             }
3730087           else
3730088             {
3730089               days += 365;
3730090             }
3730091         }
3730092       //
3730093       // After all, must subtract a day from the total.
3730094       //
3730095       days--;
3730096       //
3730097       timer_aux    = days;
3730098       timer_aux   *= 24;
3730099       timer_aux   *= (60 * 60);
3730100       timer_total += timer_aux;
3730101       //
3730102       // That's all.
3730103       //
3730104       return (timer_total);
3730105   }
3730106   //----------------------------------------------------------------------
3730107   int
3730108   leap_year (int year)
3730109   {
3730110       if ((year % 4) == 0)
3730111         {
3730112           if ((year % 100) == 0)
3730113             {
3730114               if ((year % 400) == 0)
3730115                 {
3730116                   return (1);
3730117                 }
```

```
3730118             else
3730119               {
3730120                 return (0);
3730121               }
3730122           }
3730123         else
3730124           {
3730125             return (1);
3730126           }
3730127       }
3730128     else
3730129       {
3730130         return (0);
3730131       }
3730132 }
```

101.16.5 lib/time/stime.c

«

Si veda la sezione 93.39.

```
3740001 #include <time.h>
3740002 #include <sys/os16.h>
3740003 //----------------------------------------------------------------------
3740004 int
3740005 stime (time_t *timer)
3740006 {
3740007     sysmsg_stime_t msg;
3740008     msg.timer = *timer;
3740009     msg.ret = 0;
3740010     sys (SYS_STIME, &msg, (sizeof msg));
3740011     return (msg.ret);
3740012 }
```

101.16.6 lib/time/time.c

«

Si veda la sezione 93.39.

```
3750001 #include <time.h>
3750002 #include <sys/os16.h>
3750003 //----------------------------------------------------------------------
3750004 time_t
3750005 time (time_t *timer)
3750006 {
3750007     sysmsg_time_t msg;
3750008     msg.ret = ((time_t) 0);
3750009     sys (SYS_TIME, &msg, (sizeof msg));
3750010     if (timer != NULL)
```

```
3750011         {
3750012           *timer = msg.ret;
3750013         }
3750014       return (msg.ret);
3750015   }
```

101.17 os16: «lib/unistd.h»

«

Si veda la sezione 97.2.

```
3760001 #ifndef _UNISTD_H
3760002 #define _UNISTD_H       1
3760003 
3760004 #include <const.h>
3760005 #include <sys/stat.h>
3760006 #include <sys/os16.h>
3760007 #include <sys/types.h>  // size_t, ssize_t, uid_t, gid_t, off_t, pid_t
3760008 #include <inttypes.h>   // intptr_t
3760009 #include <SEEK.h>       // SEEK_CUR, SEEK_SET, SEEK_END
3760010 //----------------------------------------------------------------------
3760011 extern char **environ;  // Variable `environ' is used by functions like
3760012                         // `execv()' in replacement for `envp[][]'.
3760013 //----------------------------------------------------------------------
3760014 extern char *optarg;    // Used by `optarg()'.
3760015 extern int   optind;    //
3760016 extern int   opterr;    //
3760017 extern int   optopt;    //
3760018 //----------------------------------------------------------------------
3760019 #define STDIN_FILENO    0       //
3760020 #define STDOUT_FILENO   1       // Standard file descriptors.
3760021 #define STDERR_FILENO   2       //
3760022 //----------------------------------------------------------------------
3760023 #define R_OK            4       // Read permission.
3760024 #define W_OK            2       // Write permission.
3760025 #define X_OK            1       // Execute or traverse permission.
3760026 #define F_OK            0       // File exists.
3760027 //----------------------------------------------------------------------
3760028 
3760029 int          access      (const char *path, int mode);
3760030 int          chdir       (const char *path);
3760031 int          chown       (const char *path, uid_t uid, gid_t gid);
3760032 int          close       (int fdn);
3760033 int          dup         (int fdn_old);
3760034 int          dup2        (int fdn_old, int fdn_new);
3760035 int          execl       (const char *path, const char *arg, ...);
3760036 int          execle      (const char *path, const char *arg, ...);
3760037 int          execlp      (const char *path, const char *arg, ...);
3760038 int          execv       (const char *path, char *const argv[]);
```

```
3760039 int            execve      (const char *path, char *const argv[],
3760040                             char *const envp[]);
3760041 int            execvp      (const char *path, char *const argv[]);
3760042 void          _exit        (int status);
3760043 int            fchown      (int fdn, uid_t uid, gid_t gid);
3760044 pid_t          fork        (void);
3760045 char          *getcwd      (char *buffer, size_t size);
3760046 uid_t          geteuid     (void);
3760047 int            getopt      (int argc, char *const argv[],
3760048                             const char *optstring);
3760049 pid_t          getpgrp     (void);
3760050 pid_t          getppid     (void);
3760051 pid_t          getpid      (void);
3760052 uid_t          getuid      (void);
3760053 int            isatty      (int fdn);
3760054 int            link        (const char *path_old, const char *path_new);
3760055 off_t          lseek       (int fdn, off_t offset, int whence);
3760056 #define        nice(n)     (0)
3760057 ssize_t        read        (int fdn, void *buffer, size_t count);
3760058 #define        readlink(p,b,s) ((ssize_t) -1)
3760059 int            rmdir       (const char *path);
3760060 int            seteuid     (uid_t uid);
3760061 int            setpgrp     (void);
3760062 int            setuid      (uid_t uid);
3760063 unsigned int sleep         (unsigned int s);
3760064 #define        sync()      /**/
3760065 char          *ttyname     (int fdn);
3760066 int            unlink      (const char *path);
3760067 ssize_t        write       (int fdn, const void *buffer, size_t count);
3760068
3760069 #endif
```

101.17.1 lib/unistd/_exit.c

Si veda la sezione 93.2.

```
3770001 #include <unistd.h>
3770002 #include <sys/os16.h>
3770003 //----------------------------------------------------------------------
3770004 void
3770005 _exit (int status)
3770006 {
3770007     sysmsg_exit_t msg;
3770008     //
3770009     // Only the low eight bit are returned.
3770010     //
3770011     msg.status = (status & 0xFF);
3770012     //
```

```
3770013     //
3770014     //
3770015     sys (SYS_EXIT, &msg, (sizeof msg));
3770016     //
3770017     // Should not return from system call, but if it does, loop
3770018     // forever:
3770019     //
3770020     while (1);
3770021 }
```

101.17.2 lib/unistd/access.c

Si veda la sezione 94.1.

```
3780001 #include <unistd.h>
3780002 #include <sys/stat.h>
3780003 #include <errno.h>
3780004 //----------------------------------------------------------------------
3780005 int
3780006 access (const char *path, int mode)
3780007 {
3780008     struct stat st;
3780009     int         status;
3780010     uid_t       euid;
3780011     //
3780012     status = stat (path, &st);
3780013     if (status != 0)
3780014       {
3780015         return (-1);
3780016       }
3780017     //
3780018     // File exists?
3780019     //
3780020     if (mode == F_OK)
3780021       {
3780022         return (0);
3780023       }
3780024     //
3780025     // Some access permissions are requested: get effective user id.
3780026     //
3780027     euid = geteuid ();
3780028     //
3780029     // Check owner access permissions.
3780030     //
3780031     if (st.st_uid == euid && ((st.st_mode & S_IRWXU) == (mode << 6)))
3780032       {
3780033         return (0);
3780034       }
```

```
3780035      //
3780036      // Check others access permissions.
3780037      //
3780038      if ((st.st_mode & S_IRWXO) == (mode))
3780039        {
3780040          return (0);
3780041        }
3780042      //
3780043      // Otherwise there are no access permissions.
3780044      //
3780045      errset (EACCES);                        // Permission denied.
3780046      return (-1);
3780047  }
```

101.17.3 lib/unistd/chdir.c

«

Si veda la sezione 93.3.

```
3790001  #include <unistd.h>
3790002  #include <string.h>
3790003  #include <const.h>
3790004  #include <sys/os16.h>
3790005  #include <errno.h>
3790006  #include <limits.h>
3790007  //----------------------------------------------------------------------
3790008  int
3790009  chdir (const char *path)
3790010  {
3790011      sysmsg_chdir_t msg;
3790012      //
3790013      msg.ret     = 0;
3790014      msg.errno   = 0;
3790015      //
3790016      strncpy (msg.path, path, PATH_MAX);
3790017      //
3790018      sys (SYS_CHDIR, &msg, (sizeof msg));
3790019      //
3790020      errno = msg.errno;
3790021      errln = msg.errln;
3790022      strncpy (errfn, msg.errfn, PATH_MAX);
3790023      return (msg.ret);
3790024  }
```

101.17.4 lib/unistd/chown.c

Si veda la sezione 93.5.

```
3800001 #include <unistd.h>
3800002 #include <string.h>
3800003 #include <const.h>
3800004 #include <sys/os16.h>
3800005 #include <errno.h>
3800006 #include <limits.h>
3800007 //----------------------------------------------------------------------
3800008 int
3800009 chown (const char *path, uid_t uid, gid_t gid)
3800010 {
3800011     sysmsg_chown_t msg;
3800012     //
3800013     strncpy (msg.path, path, PATH_MAX);
3800014     msg.uid = uid;
3800015     msg.gid = gid;
3800016     //
3800017     sys (SYS_CHOWN, &msg, (sizeof msg));
3800018     //
3800019     errno = msg.errno;
3800020     errln = msg.errln;
3800021     strncpy (errfn, msg.errfn, PATH_MAX);
3800022     return (msg.ret);
3800023 }
```

101.17.5 lib/unistd/close.c

Si veda la sezione 93.7.

```
3810001 #include <unistd.h>
3810002 #include <errno.h>
3810003 #include <sys/os16.h>
3810004 #include <string.h>
3810005 //----------------------------------------------------------------------
3810006 int
3810007 close (int fdn)
3810008 {
3810009     sysmsg_close_t msg;
3810010     msg.fdn = fdn;
3810011     sys (SYS_CLOSE, &msg, (sizeof msg));
3810012     errno = msg.errno;
3810013     errln = msg.errln;
3810014     strncpy (errfn, msg.errfn, PATH_MAX);
3810015     return (msg.ret);
3810016 }
```

101.17.6 lib/unistd/dup.c

Si veda la sezione 93.8.

```
3820001 #include <unistd.h>
3820002 #include <sys/os16.h>
3820003 #include <string.h>
3820004 #include <errno.h>
3820005 //----------------------------------------------------------------------
3820006 int
3820007 dup (int fdn_old)
3820008 {
3820009     sysmsg_dup_t msg;
3820010     //
3820011     msg.fdn_old = fdn_old;
3820012     //
3820013     sys (SYS_DUP, &msg, (sizeof msg));
3820014     //
3820015     errno = msg.errno;
3820016     errln = msg.errln;
3820017     strncpy (errfn, msg.errfn, PATH_MAX);
3820018     return (msg.ret);
3820019 }
```

101.17.7 lib/unistd/dup2.c

Si veda la sezione 93.8.

```
3830001 #include <unistd.h>
3830002 #include <sys/os16.h>
3830003 #include <string.h>
3830004 #include <errno.h>
3830005 //----------------------------------------------------------------------
3830006 int
3830007 dup2 (int fdn_old, int fdn_new)
3830008 {
3830009     sysmsg_dup2_t msg;
3830010     //
3830011     msg.fdn_old = fdn_old;
3830012     msg.fdn_new = fdn_new;
3830013     //
3830014     sys (SYS_DUP2, &msg, (sizeof msg));
3830015     //
3830016     errno = msg.errno;
3830017     errln = msg.errln;
3830018     strncpy (errfn, msg.errfn, PATH_MAX);
3830019     return (msg.ret);
3830020 }
```

101.17.8 lib/unistd/environ.c

Si veda la sezione 97.1.

```
3840001 #include <unistd.h>
3840002 //----------------------------------------------------------------------
3840003 char **environ;
```

101.17.9 lib/unistd/execl.c

Si veda la sezione 94.20.

```
3850001 #include <unistd.h>
3850002 //----------------------------------------------------------------------
3850003 int
3850004 execl (const char *path, const char *arg, ...)
3850005 {
3850006     int   argc;
3850007     char *arg_next;
3850008     char *argv[ARG_MAX/2];
3850009     //
3850010     va_list ap;
3850011     va_start (ap, arg);
3850012     //
3850013     arg_next = arg;
3850014     //
3850015     for (argc = 0; argc < ARG_MAX/2; argc++)
3850016       {
3850017         argv[argc] = arg_next;
3850018         if (argv[argc] == NULL)
3850019           {
3850020             break;      // End of arguments.
3850021           }
3850022         arg_next = va_arg (ap, char *);
3850023       }
3850024     //
3850025     return (execve (path, argv, environ));      // [1]
3850026 }
3850027 //
3850028 // The variable 'environ' is declared as 'char **environ' and is
3850029 // included from <unistd.h>.
3850030 //
```

101.17.10 lib/unistd/execle.c

«

Si veda la sezione 94.20.

```
3860001 #include <unistd.h>
3860002 //----------------------------------------------------------------------
3860003 int
3860004 execle (const char *path, const char *arg, ...)
3860005 {
3860006     int    argc;
3860007     char  *arg_next;
3860008     char  *argv[ARG_MAX/2];
3860009     char **envp;
3860010     //
3860011     va_list ap;
3860012     va_start (ap, arg);
3860013     //
3860014     arg_next = arg;
3860015     //
3860016     for (argc = 0; argc < ARG_MAX/2; argc++)
3860017       {
3860018         argv[argc] = arg_next;
3860019         if (argv[argc] == NULL)
3860020           {
3860021             break;      // End of arguments.
3860022           }
3860023         arg_next = va_arg (ap, char *);
3860024       }
3860025     //
3860026     envp = va_arg (ap, char **);
3860027     //
3860028     return (execve (path, argv, envp));
3860029 }
```

101.17.11 lib/unistd/execlp.c

«

Si veda la sezione 94.20.

```
3870001 #include <unistd.h>
3870002 #include <string.h>
3870003 #include <stdlib.h>
3870004 #include <errno.h>
3870005 #include <sys/os16.h>
3870006 //----------------------------------------------------------------------
3870007 int
3870008 execlp (const char *path, const char *arg, ...)
3870009 {
3870010     int   argc;
3870011     char *arg_next;
```

```
3870012     char *argv[ARG_MAX/2];
3870013     char  command[PATH_MAX];
3870014     int   status;
3870015     //
3870016     va_list ap;
3870017     va_start (ap, arg);
3870018     //
3870019     arg_next = arg;
3870020     //
3870021     for (argc = 0; argc < ARG_MAX/2; argc++)
3870022       {
3870023         argv[argc] = arg_next;
3870024         if (argv[argc] == NULL)
3870025           {
3870026             break;      // End of arguments.
3870027           }
3870028         arg_next = va_arg (ap, char *);
3870029       }
3870030     //
3870031     // Get a full command path if necessary.
3870032     //
3870033     status = namep (path, command, (size_t) PATH_MAX);
3870034     if (status != 0)
3870035       {
3870036         //
3870037         // Variable 'errno' is already set by 'commandp()'.
3870038         //
3870039         return (-1);
3870040       }
3870041     //
3870042     // Return calling 'execve()'
3870043     //
3870044     return (execve (command, argv, environ));   // [1]
3870045 }
3870046 //
3870047 // The variable 'environ' is declared as 'char **environ' and is
3870048 // included from <unistd.h>.
3870049 //
```

101.17.12 lib/unistd/execv.c

Si veda la sezione 94.20.

```
3880001 #include <unistd.h>
3880002 //----------------------------------------------------------------------
3880003 int
3880004 execv (const char *path, char *const argv[])
3880005 {
```

```
3880006     return (execve (path, argv, environ));      // [1]
3880007 }
3880008 //
3880009 // The variable 'environ' is declared as 'char **environ' and is
3880010 // included from <unistd.h>.
3880011 //
```

101.17.13 lib/unistd/execve.c

Si veda la sezione 93.10.

```
3890001 #include <unistd.h>
3890002 #include <sys/types.h>
3890003 #include <sys/os16.h>
3890004 #include <errno.h>
3890005 #include <string.h>
3890006 #include <string.h>
3890007 //----------------------------------------------------------------------
3890008 int
3890009 execve (const char *path, char *const argv[], char *const envp[])
3890010 {
3890011     sysmsg_exec_t msg;
3890012     size_t          size;
3890013     size_t          arg_size;
3890014     int             argc;
3890015     size_t          env_size;
3890016     int             envc;
3890017     char           *arg_data = msg.arg_data;
3890018     char           *env_data = msg.env_data;
3890019     //
3890020     msg.ret   = 0;
3890021     msg.errno = 0;
3890022     //
3890023     strncpy (msg.path, path, PATH_MAX);
3890024     //
3890025     // Copy 'argv[]' inside a the message buffer 'msg.arg_data',
3890026     // separating each string with a null character and counting the
3890027     // number of strings inside 'argc'.
3890028     //
3890029     for (argc = 0, arg_size = 0, size = 0;
3890030          argv       != NULL          &&
3890031          argc        < (ARG_MAX/16)  &&
3890032          arg_size    < ARG_MAX/2     &&
3890033          argv[argc] != NULL;
3890034          argc++, arg_size += size)
3890035       {
3890036         size = strlen (argv[argc]);
3890037         size++;          // Count also the final null character.
```

```
3890038         if (size > (ARG_MAX/2 - arg_size))
3890039           {
3890040             errset (E2BIG);      // Argument list too long.
3890041             return (-1);
3890042           }
3890043         strncpy (arg_data, argv[argc], size);
3890044         arg_data += size;
3890045       }
3890046     msg.argc = argc;
3890047     //
3890048     // Copy 'envp[]' inside a the message buffer 'msg.env_data',
3890049     // separating each string with a null character and counting the
3890050     // number of strings inside 'envc'.
3890051     //
3890052     for (envc = 0, env_size = 0, size = 0;
3890053          envp       != NULL         &&
3890054          envc        < (ARG_MAX/16) &&
3890055          env_size    < ARG_MAX/2    &&
3890056          envp[envc] != NULL;
3890057          envc++, env_size += size)
3890058       {
3890059         size = strlen (envp[envc]);
3890060         size++;          // Count also the final null character.
3890061         if (size > (ARG_MAX/2 - env_size))
3890062           {
3890063             errset (E2BIG);      // Argument list too long.
3890064             return (-1);
3890065           }
3890066         strncpy (env_data, envp[envc], size);
3890067         env_data += size;
3890068       }
3890069     msg.envc = envc;
3890070     //
3890071     // System call.
3890072     //
3890073     sys (SYS_EXEC, &msg, (sizeof msg));
3890074     //
3890075     // Should not return, but if it does, then there is an error.
3890076     //
3890077     errno = msg.errno;
3890078     errln = msg.errln;
3890079     strncpy (errfn, msg.errfn, PATH_MAX);
3890080     return (msg.ret);
3890081 }
```

101.17.14 lib/unistd/execvp.c

«

Si veda la sezione 94.20.

```
3900001 #include <unistd.h>
3900002 #include <string.h>
3900003 #include <stdlib.h>
3900004 #include <errno.h>
3900005 #include <sys/os16.h>
3900006 //----------------------------------------------------------------------
3900007 int
3900008 execvp (const char *path, char *const argv[])
3900009 {
3900010     char  command[PATH_MAX];
3900011     int   status;
3900012     //
3900013     // Get a full command path if necessary.
3900014     //
3900015     status = namep (path, command, (size_t) PATH_MAX);
3900016     if (status != 0)
3900017       {
3900018         //
3900019         // Variable 'errno' is already set by 'namep()'.
3900020         //
3900021         return (-1);
3900022       }
3900023     //
3900024     // Return calling 'execve()'
3900025     //
3900026     return (execve (command, argv, environ));   // [1]
3900027 }
3900028 //
3900029 // The variable 'environ' is declared as 'char **environ' and is
3900030 // included from <unistd.h>.
3900031 //
```

101.17.15 lib/unistd/fchdir.c

«

Si veda la sezione 97.2.

```
3910001 #include <unistd.h>
3910002 #include <errno.h>
3910003 //----------------------------------------------------------------------
3910004 int
3910005 fchdir (int fdn)
3910006 {
3910007     //
3910008     // os16 requires to keep track of the path for the current working
3910009     // directory. The standard function 'fchdir()' is not applicable.
```

```
3910010     //
3910011     errset (E_NOT_IMPLEMENTED);
3910012     return (-1);
3910013 }
```

101.17.16 lib/unistd/fchown.c

Si veda la sezione 93.5.

```
3920001 #include <unistd.h>
3920002 #include <string.h>
3920003 #include <const.h>
3920004 #include <sys/os16.h>
3920005 #include <errno.h>
3920006 #include <limits.h>
3920007 //----------------------------------------------------------------------
3920008 int
3920009 fchown (int fdn, uid_t uid, gid_t gid)
3920010 {
3920011     sysmsg_fchown_t msg;
3920012     //
3920013     msg.fdn = fdn;
3920014     msg.uid = uid;
3920015     msg.gid = gid;
3920016     //
3920017     sys (SYS_FCHOWN, &msg, (sizeof msg));
3920018     //
3920019     errno = msg.errno;
3920020     errln = msg.errln;
3920021     strncpy (errfn, msg.errfn, PATH_MAX);
3920022     return (msg.ret);
3920023 }
```

101.17.17 lib/unistd/fork.c

Si veda la sezione 93.14.

```
3930001 #include <unistd.h>
3930002 #include <sys/types.h>
3930003 #include <sys/os16.h>
3930004 #include <errno.h>
3930005 #include <string.h>
3930006 //----------------------------------------------------------------------
3930007 pid_t
3930008 fork (void)
3930009 {
3930010     sysmsg_fork_t msg;
```

```
3930011     //
3930012     // Set the return value for the child process.
3930013     //
3930014     msg.ret = 0;
3930015     //
3930016     // Do the system call.
3930017     //
3930018     sys (SYS_FORK, &msg, (sizeof msg));
3930019     //
3930020     // If the system call has successfully generated a copy of
3930021     // the original process, the following code is executed from
3930022     // the parent and the child. But the child has the 'msg'
3930023     // structure untouched, while the parent has, at least, the
3930024     // pid number inside 'msg.ret'.
3930025     // If the system call fails, there is no child, and the
3930026     // parent finds the return value equal to -1, with an
3930027     // error number.
3930028     //
3930029     errno = msg.errno;
3930030     errln = msg.errln;
3930031     strncpy (errfn, msg.errfn, PATH_MAX);
3930032     return (msg.ret);
3930033 }
```

101.17.18 lib/unistd/getcwd.c

«

Si veda la sezione 93.16.

```
3940001 #include <unistd.h>
3940002 #include <sys/types.h>
3940003 #include <sys/os16.h>
3940004 #include <errno.h>
3940005 #include <stddef.h>
3940006 #include <string.h>
3940007 //----------------------------------------------------------------------
3940008 char *
3940009 getcwd (char *buffer, size_t size)
3940010 {
3940011     sysmsg_uarea_t msg;
3940012     //
3940013     // Check arguments: the buffer must be given.
3940014     //
3940015     if (buffer == NULL)
3940016       {
3940017         errset (EINVAL);
3940018         return (NULL);
3940019       }
3940020     //
```

```
3940021     // Make shure that the last character, inside the working directory
3940022     // path is a null character.
3940023     //
3940024     msg.path_cwd[PATH_MAX-1] = 0;
3940025     //
3940026     // Just get the user area data.
3940027     //
3940028     sys (SYS_UAREA, &msg, (sizeof msg));
3940029     //
3940030     // Check that the path is still correctly terminated. If it isn't,
3940031     // the path is longer than the implementation limits, and it is
3940032     // really *bad*.
3940033     //
3940034     if (msg.path_cwd[PATH_MAX-1] != 0)
3940035       {
3940036         errset (E_LIMIT);       // Exceeded implementation limits.
3940037         return (NULL);
3940038       }
3940039     //
3940040     // If the path is larger than the buffer size, return an error.
3940041     // Please note that the parameter 'size' must include the
3940042     // terminating null character, so, if the string is equal to
3940043     // the size, it is already beyond the size limit.
3940044     //
3940045     if (strlen (msg.path_cwd) >= size)
3940046       {
3940047         errset (ERANGE);        // Result too large.
3940048         return (NULL);
3940049       }
3940050     //
3940051     // Everything is fine, so, copy the path to the buffer and return.
3940052     //
3940053     strncpy (buffer, msg.path_cwd, size);
3940054     return (buffer);
3940055 }
```

101.17.19 lib/unistd/geteuid.c

Si veda la sezione 93.18.

```
3950001 #include <unistd.h>
3950002 #include <sys/types.h>
3950003 #include <sys/os16.h>
3950004 #include <errno.h>
3950005 //----------------------------------------------------------------------
3950006 uid_t
3950007 geteuid (void)
3950008 {
```

```
3950009     sysmsg_uarea_t msg;
3950010     sys (SYS_UAREA, &msg, (sizeof msg));
3950011     return (msg.euid);
3950012 }
```

101.17.20 lib/unistd/getopt.c

Si veda la sezione 94.52.

```
3960001 #include <unistd.h>
3960002 #include <sys/types.h>
3960003 #include <sys/os16.h>
3960004 #include <errno.h>
3960005 //----------------------------------------------------------------------
3960006 char *optarg;
3960007 int   optind    = 1;
3960008 int   opterr    = 1;
3960009 int   optopt    = 0;
3960010 //----------------------------------------------------------------------
3960011 static void getopt_no_argument (int opt);
3960012 //----------------------------------------------------------------------
3960013 int
3960014 getopt (int argc, char *const argv[], const char *optstring)
3960015 {
3960016     static int o = 0;           // Index to scan grouped options.
3960017            int s;               // Index to scan 'optstring'
3960018            int opt;             // Current option letter.
3960019            int flag_argument;   // If there should be an argument.
3960020     //
3960021     // Entering the function, 'flag_argument' is zero. Just to make
3960022     // it clear:
3960023     //
3960024     flag_argument = 0;
3960025     //
3960026     // Scan 'argv[]' elements, starting form the value that 'optind'
3960027     // already have.
3960028     //
3960029     for (; optind < argc; optind++)
3960030       {
3960031         //
3960032         // If an option is expected, some check must be done at
3960033         // the beginning.
3960034         //
3960035         if (!flag_argument)
3960036           {
3960037             //
3960038             // Check if the scan is finished and 'optind' should be kept
3960039             // untouched:
```

```
3960040                 //    'argv[optind]'    is a null pointer;
3960041                 //    'argv[optind][0]' is not the character '-';
3960042                 //    'argv[optind]'    points to the string "-";
3960043                 //   all 'argv[]' elements are parsed.
3960044                 //
3960045                 if (argv[optind] == NULL
3960046                  || argv[optind][0] != '-'
3960047                  || argv[optind][1] == 0
3960048                  || optind >= argc)
3960049                   {
3960050                     return (-1);
3960051                   }
3960052                 //
3960053                 // Check if the scan is finished and 'optind' is to be
3960054                 // incremented:
3960055                 //    'argv[optind]'    points to the string "--".
3960056                 //
3960057                 if (argv[optind][0] == '-'
3960058                  && argv[optind][1] == '-'
3960059                  && argv[optind][2] == 0)
3960060                   {
3960061                     optind++;
3960062                     return (-1);
3960063                   }
3960064               }
3960065             //
3960066             // Scan 'argv[optind]' using the static index 'o'.
3960067             //
3960068             for (; o < strlen (argv[optind]); o++)
3960069               {
3960070                 //
3960071                 // If there should be an option, index 'o' should
3960072                 // start from 1, because 'argv[optind][0]' must
3960073                 // be equal to '-'.
3960074                 //
3960075                 if (!flag_argument && (o == 0))
3960076                   {
3960077                     //
3960078                     // As there is no options, 'o' cannot start
3960079                     // from zero, so a new loop is done.
3960080                     //
3960081                     continue;
3960082                   }
3960083                 //
3960084                 if (flag_argument)
3960085                   {
3960086                     //
3960087                     // There should be an argument, starting from
```

```
3960088                     // `argv[optind][o]'.
3960089                     //
3960090                     if ((o == 0) && (argv[optind][o] == '-'))
3960091                       {
3960092                         //
3960093                         // `argv[optind][0]' is equal to '-', but there
3960094                         // should be an argument instead: the argument
3960095                         // is missing.
3960096                         //
3960097                         optarg = NULL;
3960098                         //
3960099                         if (optstring[0] == ':')
3960100                           {
3960101                             //
3960102                             // As the option string starts with `:' the
3960103                             // function must return `:'.
3960104                             //
3960105                             optopt = opt;
3960106                             opt = ':';
3960107                           }
3960108                         else
3960109                           {
3960110                             //
3960111                             // As the option string does not start with `:'
3960112                             // the function must return `?'.
3960113                             //
3960114                             getopt_no_argument (opt);
3960115                             optopt = opt;
3960116                             opt = '?';
3960117                           }
3960118                         //
3960119                         // `optind' is left untouched.
3960120                         //
3960121                       }
3960122                     else
3960123                       {
3960124                         //
3960125                         // The argument is found: `optind' is to be
3960126                         // incremented and `o' is reset.
3960127                         //
3960128                         optarg = &argv[optind][o];
3960129                         optind++;
3960130                         o = 0;
3960131                       }
3960132                     //
3960133                     // Return the option, or ':', or '?'.
3960134                     //
3960135                     return (opt);
```

```
3960136        }
3960137       else
3960138        {
3960139         //
3960140         // It should be an option: ‘optstring[]’ must be
3960141         // scanned.
3960142         //
3960143         opt = argv[optind][o];
3960144         //
3960145         for (s = 0, optopt = 0; s < strlen (optstring); s++)
3960146          {
3960147           //
3960148           // If ‘optsting[0]’ is equal to ’:’, index ‘s’ must
3960149           // start at 1.
3960150           //
3960151           if ((s == 0) && (optstring[0] == ’:’))
3960152            {
3960153             continue;
3960154            }
3960155           //
3960156           if (opt == optstring[s])
3960157            {
3960158             //
3960159             if (optstring[s+1] == ’:’)
3960160              {
3960161               //
3960162               // There is an argument.
3960163               //
3960164               flag_argument = 1;
3960165               break;
3960166              }
3960167             else
3960168              {
3960169               //
3960170               // There is no argument.
3960171               //
3960172               o++;
3960173               return (opt);
3960174              }
3960175            }
3960176          }
3960177         //
3960178         if (s >= strlen (optstring))
3960179          {
3960180           //
3960181           // The ‘optstring’ scan is concluded with no
3960182           // match.
3960183           //
```

```
3960184                     o++;
3960185                     optopt = opt;
3960186                     return ('?');
3960187                   }
3960188                 //
3960189                 // Otherwise the loop was broken.
3960190                 //
3960191               }
3960192           }
3960193         //
3960194         // Check index 'o'.
3960195         //
3960196         if (o >= strlen (argv[optind]))
3960197           {
3960198             //
3960199             // There are no more options or there is no argument
3960200             // inside current 'argv[optind]' string. Index 'o' is
3960201             // reset before the next loop.
3960202             //
3960203             o = 0;
3960204           }
3960205       }
3960206     //
3960207     // No more elements inside 'argv' or loop broken: there might be a
3960208     // missing argument.
3960209     //
3960210     if (flag_argument)
3960211       {
3960212         //
3960213         // Missing option argument.
3960214         //
3960215         optarg = NULL;
3960216         //
3960217         if (optstring[0] == ':')
3960218           {
3960219             return (':');
3960220           }
3960221         else
3960222           {
3960223             getopt_no_argument (opt);
3960224             return ('?');
3960225           }
3960226       }
3960227     //
3960228     return (-1);
3960229 }
3960230 //----------------------------------------------------------------------
3960231 static void
```

```
3960232 getopt_no_argument (int opt)
3960233 {
3960234     if (opterr)
3960235       {
3960236         fprintf (stderr, "Missing argument for option `-%c'\n", opt);
3960237       }
3960238 }
```

101.17.21 lib/unistd/getpgrp.c

Si veda la sezione 93.20.

```
3970001 #include <unistd.h>
3970002 #include <sys/types.h>
3970003 #include <sys/os16.h>
3970004 #include <errno.h>
3970005 //----------------------------------------------------------------------
3970006 pid_t
3970007 getpgrp (void)
3970008 {
3970009     sysmsg_uarea_t msg;
3970010     sys (SYS_UAREA, &msg, (sizeof msg));
3970011     return (msg.pgrp);
3970012 }
```

101.17.22 lib/unistd/getpid.c

Si veda la sezione 93.20.

```
3980001 #include <unistd.h>
3980002 #include <sys/types.h>
3980003 #include <sys/os16.h>
3980004 #include <errno.h>
3980005 //----------------------------------------------------------------------
3980006 pid_t
3980007 getpid (void)
3980008 {
3980009     sysmsg_uarea_t msg;
3980010     sys (SYS_UAREA, &msg, (sizeof msg));
3980011     return (msg.pid);
3980012 }
```

101.17.23 lib/unistd/getppid.c

Si veda la sezione 93.20.

```
3990001 #include <unistd.h>
3990002 #include <sys/types.h>
3990003 #include <sys/os16.h>
3990004 #include <errno.h>
3990005 //----------------------------------------------------------------------
3990006 pid_t
3990007 getppid (void)
3990008 {
3990009     sysmsg_uarea_t msg;
3990010     sys (SYS_UAREA, &msg, (sizeof msg));
3990011     return (msg.ppid);
3990012 }
```

101.17.24 lib/unistd/getuid.c

Si veda la sezione 93.18.

```
4000001 #include <unistd.h>
4000002 #include <sys/types.h>
4000003 #include <sys/os16.h>
4000004 #include <errno.h>
4000005 //----------------------------------------------------------------------
4000006 uid_t
4000007 getuid (void)
4000008 {
4000009     sysmsg_uarea_t msg;
4000010     sys (SYS_UAREA, &msg, (sizeof msg));
4000011     return (msg.uid);
4000012 }
```

101.17.25 lib/unistd/isatty.c

Si veda la sezione 94.61.

```
4010001 #include <sys/stat.h>
4010002 #include <sys/os16.h>
4010003 #include <unistd.h>
4010004 #include <sys/types.h>
4010005 #include <errno.h>
4010006 //----------------------------------------------------------------------
4010007 int
4010008 isatty (int fdn)
4010009 {
4010010     struct stat file_status;
```

```
4010011     //
4010012     // Verify to have valid input data.
4010013     //
4010014     if (fdn < 0)
4010015       {
4010016         errset (EBADF);
4010017         return (0);
4010018       }
4010019     //
4010020     // Verify the standard input.
4010021     //
4010022     if (fstat(fdn, &file_status) == 0)
4010023       {
4010024         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
4010025           {
4010026             return (1); // Meaning it is ok!
4010027           }
4010028         if (major (file_status.st_rdev) == DEV_TTY_MAJOR)
4010029           {
4010030             return (1); // Meaning it is ok!
4010031           }
4010032       }
4010033     else
4010034       {
4010035         errset (errno);
4010036         return (0);
4010037       }
4010038     //
4010039     // If here, it is not a terminal of any kind.
4010040     //
4010041     errset (EINVAL);
4010042     return (0);
4010043 }
```

101.17.26 lib/unistd/link.c

Si veda la sezione 93.23.

```
4020001 #include <unistd.h>
4020002 #include <string.h>
4020003 #include <const.h>
4020004 #include <sys/os16.h>
4020005 #include <errno.h>
4020006 #include <limits.h>
4020007 //----------------------------------------------------------------------
4020008 int
4020009 link (const char *path_old, const char *path_new)
4020010 {
```

```
4020011     sysmsg_link_t msg;
4020012     //
4020013     strncpy (msg.path_old, path_old, PATH_MAX);
4020014     strncpy (msg.path_new, path_new, PATH_MAX);
4020015     //
4020016     sys (SYS_LINK, &msg, (sizeof msg));
4020017     //
4020018     errno = msg.errno;
4020019     errln = msg.errln;
4020020     strncpy (errfn, msg.errfn, PATH_MAX);
4020021     return (msg.ret);
4020022 }
```

101.17.27 lib/unistd/lseek.c

«

Si veda la sezione 93.24.

```
4030001 #include <unistd.h>
4030002 #include <sys/types.h>
4030003 #include <sys/os16.h>
4030004 #include <errno.h>
4030005 #include <string.h>
4030006 //----------------------------------------------------------------------
4030007 off_t
4030008 lseek (int fdn, off_t offset, int whence)
4030009 {
4030010     sysmsg_lseek_t  msg;
4030011     msg.fdn     = fdn;
4030012     msg.offset = offset;
4030013     msg.whence = whence;
4030014     sys (SYS_LSEEK, &msg, (sizeof msg));
4030015     errno = msg.errno;
4030016     errln = msg.errln;
4030017     strncpy (errfn, msg.errfn, PATH_MAX);
4030018     return (msg.ret);
4030019 }
```

101.17.28 lib/unistd/read.c

«

Si veda la sezione 93.29.

```
4040001 #include <unistd.h>
4040002 #include <sys/os16.h>
4040003 #include <errno.h>
4040004 #include <string.h>
4040005 #include <stdio.h>
4040006 //----------------------------------------------------------------------
```

```
4040007 ssize_t
4040008 read (int fdn, void *buffer, size_t count)
4040009 {
4040010     sysmsg_read_t msg;
4040011     //
4040012     // Reduce size of read if necessary.
4040013     //
4040014     if (count > BUFSIZ)
4040015       {
4040016         count = BUFSIZ;
4040017       }
4040018     //
4040019     // Fill the message.
4040020     //
4040021     msg.fdn   = fdn;
4040022     msg.count = count;
4040023     msg.eof   = 0;
4040024     msg.ret   = 0;
4040025     //
4040026     // Repeat syscall, until something is received or end of file is
4040027     // reached.
4040028     //
4040029     while (1)
4040030       {
4040031         sys (SYS_READ, &msg, (sizeof msg));
4040032         if (msg.ret != 0 || msg.eof)
4040033           {
4040034             break;
4040035           }
4040036       }
4040037     //
4040038     // Before return: be careful with the 'msg.buffer' copy, because
4040039     // it cannot be longer than 'count', otherwise, some unexpected
4040040     // memory will be overwritten!
4040041     //
4040042     if (msg.ret < 0)
4040043       {
4040044         //
4040045         // No valid read, no change inside the buffer.
4040046         //
4040047         errno = msg.errno;
4040048         return (msg.ret);
4040049       }
4040050     //
4040051     if (msg.ret > count)
4040052       {
4040053         //
4040054         // A strange value was returned. Considering it a read error.
```

```
4040055             //
4040056             errset (EIO);                       // I/O error.
4040057             return (-1);
4040058           }
4040059         //
4040060         // A valid read: fill the buffer with 'msg.ret' bytes.
4040061         //
4040062         memcpy (buffer, msg.buffer, msg.ret);
4040063         //
4040064         // Return.
4040065         //
4040066         return (msg.ret);
4040067     }
```

101.17.29 lib/unistd/rmdir.c

«

Si veda la sezione 93.30.

```
4050001 #include <unistd.h>
4050002 #include <string.h>
4050003 #include <const.h>
4050004 #include <sys/os16.h>
4050005 #include <errno.h>
4050006 #include <limits.h>
4050007 //----------------------------------------------------------------------
4050008 int
4050009 rmdir (const char *path)
4050010 {
4050011     sysmsg_stat_t   msg_stat;
4050012     sysmsg_unlink_t msg_unlink;
4050013     //
4050014     if (path == NULL)
4050015       {
4050016         errset (EINVAL);
4050017         return (-1);
4050018       }
4050019     //
4050020     strncpy (msg_stat.path, path, PATH_MAX);
4050021     //
4050022     sys (SYS_STAT, &msg_stat, (sizeof msg_stat));
4050023     //
4050024     if (msg_stat.ret != 0)
4050025       {
4050026         errno = msg_stat.errno;
4050027         errln = msg_stat.errln;
4050028         strncpy (errfn, msg_stat.errfn, PATH_MAX);
4050029         return (msg_stat.ret);
4050030       }
```

```
4050031     //
4050032     if (!S_ISDIR (msg_stat.stat.st_mode))
4050033       {
4050034         errset (ENOTDIR);              // Not a directory.
4050035         return (-1);
4050036       }
4050037     //
4050038     strncpy (msg_unlink.path, path, PATH_MAX);
4050039     //
4050040     sys (SYS_UNLINK, &msg_unlink, (sizeof msg_unlink));
4050041     //
4050042     errno = msg_unlink.errno;
4050043     errln = msg_unlink.errln;
4050044     strncpy (errfn, msg_unlink.errfn, PATH_MAX);
4050045     return (msg_unlink.ret);
4050046 }
```

101.17.30 lib/unistd/seteuid.c

Si veda la sezione 93.33.

```
4060001 #include <unistd.h>
4060002 #include <sys/types.h>
4060003 #include <sys/os16.h>
4060004 #include <errno.h>
4060005 #include <string.h>
4060006 //----------------------------------------------------------------------
4060007 int
4060008 seteuid (uid_t uid)
4060009 {
4060010     sysmsg_seteuid_t msg;
4060011     msg.ret    = 0;
4060012     msg.errno = 0;
4060013     msg.euid   = uid;
4060014     sys (SYS_SETEUID, &msg, (sizeof msg));
4060015     errno = msg.errno;
4060016     errln = msg.errln;
4060017     strncpy (errfn, msg.errfn, PATH_MAX);
4060018     return (msg.ret);
4060019 }
4060020
```

101.17.31 lib/unistd/setpgrp.c

Si veda la sezione 93.32.

```
4070001 #include <unistd.h>
4070002 #include <sys/os16.h>
4070003 #include <stddef.h>
4070004 //----------------------------------------------------------------------
4070005 int
4070006 setpgrp (void)
4070007 {
4070008     sys (SYS_PGRP, NULL, (size_t) 0);
4070009     return (0);
4070010 }
```

101.17.32 lib/unistd/setuid.c

Si veda la sezione 93.33.

```
4080001 #include <unistd.h>
4080002 #include <sys/types.h>
4080003 #include <sys/os16.h>
4080004 #include <errno.h>
4080005 #include <string.h>
4080006 //----------------------------------------------------------------------
4080007 int
4080008 setuid (uid_t uid)
4080009 {
4080010     sysmsg_setuid_t msg;
4080011     msg.ret    = 0;
4080012     msg.errno = 0;
4080013     msg.euid   = uid;
4080014     sys (SYS_SETUID, &msg, (sizeof msg));
4080015     errno = msg.errno;
4080016     errln = msg.errln;
4080017     strncpy (errfn, msg.errfn, PATH_MAX);
4080018     return (msg.ret);
4080019 }
```

101.17.33 lib/unistd/sleep.c

Si veda la sezione 93.35.

```
4090001 #include <unistd.h>
4090002 #include <sys/types.h>
4090003 #include <sys/os16.h>
4090004 #include <errno.h>
4090005 #include <time.h>
```

```
4090006 //----------------------------------------------------------------------
4090007 unsigned int
4090008 sleep (unsigned int seconds)
4090009 {
4090010     sysmsg_sleep_t  msg;
4090011     time_t          start;
4090012     time_t          end;
4090013     int             slept;
4090014     //
4090015     if (seconds == 0)
4090016       {
4090017         return (0);
4090018       }
4090019     //
4090020     msg.events  = WAKEUP_EVENT_TIMER;
4090021     msg.seconds = seconds;
4090022     sys (SYS_SLEEP, &msg, (sizeof msg));
4090023     start = msg.ret;
4090024     end   = time (NULL);
4090025     slept = end - msg.ret;
4090026     //
4090027     if (slept < 0)
4090028       {
4090029         return (seconds);
4090030       }
4090031     else if (slept < seconds)
4090032       {
4090033         return (seconds - slept);
4090034       }
4090035     else
4090036       {
4090037         return (0);
4090038       }
4090039 }
```

101.17.34 lib/unistd/ttyname.c

«

Si veda la sezione 94.124.

```
4100001 #include <sys/os16.h>
4100002 #include <sys/stat.h>
4100003 #include <unistd.h>
4100004 #include <sys/types.h>
4100005 #include <errno.h>
4100006 #include <limits.h>
4100007 //----------------------------------------------------------------------
4100008 char *
4100009 ttyname (int fdn)
```

```
4100010 {
4100011     int         dev_minor;
4100012     struct stat file_status;
4100013     static char name[PATH_MAX];
4100014     //
4100015     // Verify to have valid input data.
4100016     //
4100017     if (fdn < 0)
4100018       {
4100019         errset (EBADF);
4100020         return (NULL);
4100021       }
4100022     //
4100023     // Verify the file descriptor.
4100024     //
4100025     if (fstat (fdn, &file_status) == 0)
4100026       {
4100027         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
4100028           {
4100029             dev_minor = minor (file_status.st_rdev);
4100030             //
4100031             // If minor is equal to 0xFF, it is '/dev/console'.
4100032             //
4100033             if (dev_minor < 0xFF)
4100034               {
4100035                 sprintf (name, "/dev/console%i", dev_minor);
4100036               }
4100037             else
4100038               {
4100039                 strcpy (name, "/dev/console");
4100040               }
4100041             return (name);
4100042           }
4100043         else if (file_status.st_rdev == DEV_TTY)
4100044           {
4100045             strcpy (name, "/dev/tty");
4100046             return (name);
4100047           }
4100048         else
4100049           {
4100050             errset (ENOTTY);
4100051             return (NULL);
4100052           }
4100053       }
4100054     else
4100055       {
4100056         errset (errno);
4100057         return (NULL);
```

```
4100058         }
4100059 }
```

101.17.35 lib/unistd/unlink.c

Si veda la sezione 93.42.

```
4110001 #include <unistd.h>
4110002 #include <string.h>
4110003 #include <const.h>
4110004 #include <sys/os16.h>
4110005 #include <errno.h>
4110006 #include <limits.h>
4110007 //----------------------------------------------------------------------
4110008 int
4110009 unlink (const char *path)
4110010 {
4110011     sysmsg_unlink_t msg;
4110012     //
4110013     strncpy (msg.path, path, PATH_MAX);
4110014     //
4110015     sys (SYS_UNLINK, &msg, (sizeof msg));
4110016     //
4110017     errno = msg.errno;
4110018     errln = msg.errln;
4110019     strncpy (errfn, msg.errfn, PATH_MAX);
4110020     return (msg.ret);
4110021 }
```

101.17.36 lib/unistd/write.c

Si veda la sezione 93.44.

```
4120001 #include <unistd.h>
4120002 #include <sys/os16.h>
4120003 #include <errno.h>
4120004 #include <string.h>
4120005 #include <const.h>
4120006 #include <stdio.h>
4120007 //----------------------------------------------------------------------
4120008 ssize_t
4120009 write (int fdn, const void *buffer, size_t count)
4120010 {
4120011     sysmsg_write_t msg;
4120012     //
4120013     // Reduce size of write if necessary.
4120014     //
```

```
4120015      if (count > BUFSIZ)
4120016        {
4120017          count = BUFSIZ;
4120018        }
4120019      //
4120020      // Fill the message.
4120021      //
4120022      msg.fdn   = fdn;
4120023      msg.count = count;
4120024      memcpy (msg.buffer, buffer, count);
4120025      //
4120026      // Syscall.
4120027      //
4120028      sys (SYS_WRITE, &msg, (sizeof msg));
4120029      //
4120030      // Check result and return.
4120031      //
4120032      if (msg.ret < 0)
4120033        {
4120034          //
4120035          // No valid read, no change inside the buffer.
4120036          //
4120037          errno = msg.errno;
4120038          errln = msg.errln;
4120039          strncpy (errfn, msg.errfn, PATH_MAX);
4120040          return (msg.ret);
4120041        }
4120042      //
4120043      if (msg.ret > count)
4120044        {
4120045          //
4120046          // A strange value was returned. Considering it a read error.
4120047          //
4120048          errset (EIO);                   // I/O error.
4120049          return (-1);
4120050        }
4120051      //
4120052      // A valid write return.
4120053      //
4120054      return (msg.ret);
4120055  }
```

101.18 os16: «lib/utime.h»

Si veda la sezione 97.2.

```
4130001 #ifndef _UTIME_H
4130002 #define _UTIME_H        1
4130003
4130004 #include <const.h>
4130005 #include <restrict.h>
4130006 #include <sys/types.h>          // time_t
4130007
4130008 //----------------------------------------------------------------------
4130009 struct utimbuf {
4130010     time_t  actime;
4130011     time_t  modtime;
4130012 };
4130013 //----------------------------------------------------------------------
4130014 int utime (const char *path, const struct utimbuf *times);
4130015 //----------------------------------------------------------------------
4130016
4130017 #endif
```

101.18.1 lib/utime/utime.c

Si veda la sezione 97.2.

```
4140001 #include <utime.h>
4140002 #include <errno.h>
4140003 //----------------------------------------------------------------------
4140004 int
4140005 utime (const char *path, const struct utimbuf *times)
4140006 {
4140007     //
4140008     // Currently not implemented.
4140009     //
4140010     return (0);
4140011 }
```

Sorgenti delle applicazioni

TXT
htm
PDF
pdf
P1
P2
P3
P4
epub

102.1 os16: directory «applic/»

102.1.1 applic/MAKEDEV.c

Si veda la sezione 98.3.

```
4150001 #include <unistd.h>
4150002 #include <stdlib.h>
4150003 #include <sys/stat.h>
4150004 #include <fcntl.h>
4150005 #include <kernel/devices.h>
4150006 #include <stdio.h>
4150007 //----------------------------------------------------------------------
4150008 int
4150009 main (void)
4150010 {
4150011     int status;
4150012     status = mknod ("mem",        (mode_t) (S_IFCHR | 0444),
4150013                     (dev_t) DEV_MEM);
4150014     if (status) perror (NULL);
4150015     status = mknod ("null",       (mode_t) (S_IFCHR | 0666),
4150016                     (dev_t) DEV_NULL);
4150017     if (status) perror (NULL);
4150018     status = mknod ("port",       (mode_t) (S_IFCHR | 0644),
4150019                     (dev_t) DEV_PORT);
4150020     if (status) perror (NULL);
4150021     status = mknod ("zero",       (mode_t) (S_IFCHR | 0666),
4150022                     (dev_t) DEV_ZERO);
4150023     if (status) perror (NULL);
4150024     status = mknod ("tty",        (mode_t) (S_IFCHR | 0666),
4150025                     (dev_t) DEV_TTY);
4150026     if (status) perror (NULL);
4150027     status = mknod ("dsk0",       (mode_t) (S_IFBLK | 0644),
4150028                     (dev_t) DEV_DSK0);
4150029     if (status) perror (NULL);
4150030     status = mknod ("dsk1",       (mode_t) (S_IFBLK | 0644),
4150031                     (dev_t) DEV_DSK1);
4150032     if (status) perror (NULL);
4150033     status = mknod ("dsk2",       (mode_t) (S_IFBLK | 0644),
4150034                     (dev_t) DEV_DSK2);
4150035     if (status) perror (NULL);
4150036     status = mknod ("dsk3",       (mode_t) (S_IFBLK | 0644),
4150037                     (dev_t) DEV_DSK3);
4150038     if (status) perror (NULL);
4150039     status = mknod ("kmem_ps",    (mode_t) (S_IFCHR | 0444),
4150040                     (dev_t) DEV_KMEM_PS);
4150041     if (status) perror (NULL);
4150042     status = mknod ("kmem_mmp",   (mode_t) (S_IFCHR | 0444),
```

```
4150043                         (dev_t) DEV_KMEM_MMP);
4150044         if (status) perror (NULL);
4150045         status = mknod ("kmem_sb",     (mode_t) (S_IFCHR | 0444),
4150046                         (dev_t) DEV_KMEM_SB);
4150047         if (status) perror (NULL);
4150048         status = mknod ("kmem_inode", (mode_t) (S_IFCHR | 0444),
4150049                         (dev_t) DEV_KMEM_INODE);
4150050         if (status) perror (NULL);
4150051         status = mknod ("kmem_file",  (mode_t) (S_IFCHR | 0444),
4150052                         (dev_t) DEV_KMEM_FILE);
4150053         if (status) perror (NULL);
4150054         status = mknod ("console",     (mode_t) (S_IFCHR | 0644),
4150055                         (dev_t) DEV_CONSOLE);
4150056         if (status) perror (NULL);
4150057         status = mknod ("console0",    (mode_t) (S_IFCHR | 0644),
4150058                         (dev_t) DEV_CONSOLE0);
4150059         if (status) perror (NULL);
4150060         status = mknod ("console1",    (mode_t) (S_IFCHR | 0644),
4150061                         (dev_t) DEV_CONSOLE1);
4150062         if (status) perror (NULL);
4150063         status = mknod ("console2",    (mode_t) (S_IFCHR | 0644),
4150064                         (dev_t) DEV_CONSOLE2);
4150065         if (status) perror (NULL);
4150066         status = mknod ("console3",    (mode_t) (S_IFCHR | 0644),
4150067                         (dev_t) DEV_CONSOLE3);
4150068         if (status) perror (NULL);
4150069
4150070         return (0);
4150071     }
```

102.1.2 applic/aaa.c

Si veda la sezione 92.1.

```
4160001 #include <unistd.h>
4160002 #include <stdio.h>
4160003 //----------------------------------------------------------------------
4160004 int
4160005 main (void)
4160006 {
4160007     unsigned int count;
4160008     for (count = 0; count < 60; count++)
4160009       {
4160010         printf ("a");
4160011         sleep (1);
4160012       }
4160013     return (8);
4160014 }
```

102.1.3 applic/bbb.c

«

Si veda la sezione 92.1.

```
4170001 #include <unistd.h>
4170002 #include <stdio.h>
4170003 #include <stdlib.h>
4170004 //----------------------------------------------------------------------
4170005 int
4170006 main (void)
4170007 {
4170008     unsigned int count;
4170009     for (count = 0; count < 30; count++)
4170010       {
4170011         printf ("b");
4170012         sleep (2);
4170013       }
4170014     exit (0);
4170015     return (0);
4170016 }
```

102.1.4 applic/cat.c

«

Si veda la sezione 92.3.

```
4180001 #include <fcntl.h>
4180002 #include <sys/stat.h>
4180003 #include <stddef.h>
4180004 #include <unistd.h>
4180005 #include <stdio.h>
4180006 #include <stdlib.h>
4180007 #include <errno.h>
4180008 //----------------------------------------------------------------------
4180009 static void cat_file_descriptor (int fd);
4180010 //----------------------------------------------------------------------
4180011 int
4180012 main (int argc, char *argv[], char *envp[])
4180013 {
4180014     int         i;
4180015     int         fd;
4180016     struct stat file_status;
4180017     //
4180018     // Check if the input comes from standard input.
4180019     //
4180020     if (argc < 2)
4180021       {
4180022         cat_file_descriptor (STDIN_FILENO);
4180023         exit (0);
4180024       }
```

```
4180025      //
4180026      // There is at least an argument: scan them.
4180027      //
4180028      for(i = 1; i < argc; i++)
4180029        {
4180030          //
4180031          // Verify if the file exists.
4180032          //
4180033          if (stat(argv[i], &file_status) != 0)
4180034            {
4180035              fprintf (stderr, "File \"%s\" does not exist!\n",
4180036                       argv[i]);
4180037              continue;
4180038            }
4180039          //
4180040          // File exists: check the file type.
4180041          //
4180042          if (S_ISDIR (file_status.st_mode))
4180043            {
4180044              fprintf (stderr, "Cannot \"cat\" "
4180045                               "\"%s\": it is a directory!\n",
4180046                       argv[i]);
4180047              continue;
4180048            }
4180049          //
4180050          // File exists and can be "cat"ed.
4180051          //
4180052          fd = open (argv[i], O_RDONLY);
4180053          if (fd >= 0)
4180054            {
4180055              cat_file_descriptor (fd);
4180056              close (fd);
4180057            }
4180058          else
4180059            {
4180060              perror (NULL);
4180061              exit (1);
4180062            }
4180063        }
4180064      return (0);
4180065  }
4180066  //----------------------------------------------------------------------
4180067  static void
4180068  cat_file_descriptor (int fd)
4180069  {
4180070      ssize_t count;
4180071      char buffer[BUFSIZ];
4180072
```

```
4180073      for (;;)
4180074        {
4180075          count = read (fd, buffer, (size_t) BUFSIZ);
4180076          if (count > 0)
4180077            {
4180078              write (STDOUT_FILENO, buffer, (size_t) count);
4180079            }
4180080          else
4180081            {
4180082              break;
4180083            }
4180084        }
4180085  }
4180086
```

102.1.5 applic/ccc.c

«

Si veda la sezione 92.1.

```
4190001  #include <unistd.h>
4190002  #include <stdlib.h>
4190003  #include <signal.h>
4190004  //----------------------------------------------------------------------
4190005  int
4190006  main (void)
4190007  {
4190008      pid_t        pid;
4190009      //------------------------------------------------------------------
4190010      pid = fork ();
4190011      if (pid == 0)
4190012        {
4190013          setuid ((uid_t) 10);
4190014          execve ("/bin/aaa", NULL, NULL);
4190015          exit (0);
4190016        }
4190017      //------------------------------------------------------------------
4190018      pid = fork ();
4190019      if (pid == 0)
4190020        {
4190021          setuid ((uid_t) 11);
4190022          execve ("/bin/bbb", NULL, NULL);
4190023          exit (0);
4190024        }
4190025      //------------------------------------------------------------------
4190026      while (1)
4190027        {
4190028          ; // Just loop, to consume CPU time: it must be killed manually.
4190029        }
```

```
4190030     return (0);
4190031 }
```

102.1.6 applic/chmod.c

Si veda la sezione 92.5.

```
4200001 #include <unistd.h>
4200002 #include <stdlib.h>
4200003 #include <sys/stat.h>
4200004 #include <sys/types.h>
4200005 #include <fcntl.h>
4200006 #include <errno.h>
4200007 #include <signal.h>
4200008 #include <stdio.h>
4200009 #include <sys/wait.h>
4200010 #include <stdio.h>
4200011 #include <string.h>
4200012 #include <limits.h>
4200013 #include <sys/os16.h>
4200014 //----------------------------------------------------------------------
4200015 static void usage (void);
4200016 //----------------------------------------------------------------------
4200017 int
4200018 main (int argc, char *argv[], char *envp[])
4200019 {
4200020     int     status;
4200021     mode_t  mode;
4200022     char    *m;          // Pointer inside the octal mode string.
4200023     int     digit;
4200024     int     a;           // Argument index.
4200025     //
4200026     //
4200027     //
4200028     if (argc < 3)
4200029       {
4200030         usage ();
4200031         return (1);
4200032       }
4200033     //
4200034     // Get mode: must be the first argument.
4200035     //
4200036     for (m = argv[1]; *m != 0; m++)
4200037       {
4200038         digit = (*m - '0');
4200039         if (digit < 0 || digit > 7)
4200040           {
4200041             usage ();
```

```
4200042                 return (2);
4200043               }
4200044             mode = mode * 8 + digit;
4200045           }
4200046         //
4200047         // System call for all the remaining arguments.
4200048         //
4200049         for (a = 2; a < argc; a++)
4200050           {
4200051             status = chmod (argv[a], mode);
4200052             if (status != 0)
4200053               {
4200054                 perror (argv[a]);
4200055                 return (3);
4200056               }
4200057           }
4200058         //
4200059         // All done.
4200060         //
4200061         return (0);
4200062     }
4200063     //----------------------------------------------------------------------
4200064     static void
4200065     usage (void)
4200066     {
4200067         fprintf (stderr, "Usage:   chmod OCTAL_MODE FILE...\n");
4200068         fprintf (stderr, "Example: chmod 0640 my_file\n");
4200069     }
```

102.1.7 applic/chown.c

«

Si veda la sezione 92.6.

```
4210001     #include <unistd.h>
4210002     #include <stdlib.h>
4210003     #include <sys/stat.h>
4210004     #include <sys/types.h>
4210005     #include <fcntl.h>
4210006     #include <errno.h>
4210007     #include <stdio.h>
4210008     #include <ctype.h>
4210009     #include <pwd.h>
4210010     //----------------------------------------------------------------------
4210011     static void usage (void);
4210012     //----------------------------------------------------------------------
4210013     int
4210014     main (int argc, char *argv[], char *envp[])
4210015     {
```

```
4210016     char            *user;
4210017     int              uid;
4210018     struct passwd   *pws;
4210019     struct stat      file_status;
4210020     int              a;          // Argument index.
4210021     int              status;
4210022     //
4210023     //
4210024     //
4210025     if (argc < 3)
4210026       {
4210027         usage ();
4210028         return (1);
4210029       }
4210030     //
4210031     // Get user id number.
4210032     //
4210033     user = argv[1];
4210034     if (isdigit (*user))
4210035       {
4210036         uid = atoi (user);
4210037       }
4210038     else
4210039       {
4210040         pws = getpwnam (user);
4210041         if (pws == NULL)
4210042           {
4210043             fprintf(stderr, "Unknown user \"%s\"!\n", user);
4210044             return(2);
4210045           }
4210046         uid = pws->pw_uid;
4210047       }
4210048     //
4210049     // Now we have the user id. Start scanning file names.
4210050     //
4210051     for (a = 2; a < argc; a++)
4210052       {
4210053         //
4210054         // Verify if the file exists, through the return value of
4210055         // `stat()'. No other checks are made.
4210056         //
4210057         if (stat(argv[a], &file_status) == 0)
4210058          {
4210059             //
4210060             // Try to change ownership.
4210061             //
4210062             status = chown (argv[a], uid, file_status.st_gid);
4210063             if (status != 0)
```

```
4210064                 {
4210065                   perror (NULL);
4210066                   return (3);
4210067                 }
4210068             }
4210069           else
4210070             {
4210071               fprintf (stderr, "File \"%s\" does not exist!\n",
4210072                        argv[a]);
4210073               continue;
4210074             }
4210075         }
4210076     //
4210077     // All done.
4210078     //
4210079     return (0);
4210080 }
4210081 //----------------------------------------------------------------------
4210082 static void
4210083 usage (void)
4210084 {
4210085     fprintf (stderr, "Usage:   chown USER|UID FILE...\n");
4210086     fprintf (stderr, "Example: chown user my_file\n");
4210087 }
```

102.1.8 applic/cp.c

«

Si veda la sezione 92.7.

```
4220001 #include <sys/os16.h>
4220002 #include <sys/stat.h>
4220003 #include <sys/types.h>
4220004 #include <unistd.h>
4220005 #include <stdlib.h>
4220006 #include <fcntl.h>
4220007 #include <errno.h>
4220008 #include <signal.h>
4220009 #include <stdio.h>
4220010 #include <string.h>
4220011 #include <limits.h>
4220012 #include <libgen.h>
4220013 //----------------------------------------------------------------------
4220014 static void usage (void);
4220015 //----------------------------------------------------------------------
4220016 int
4220017 main (int argc, char *argv[], char *envp[])
4220018 {
4220019     char        *source;
```

```
4220020     char       *destination;
4220021     char       *destination_full;
4220022     struct stat file_status;
4220023     int         dest_is_a_dir = 0;
4220024     int         a;                      // Argument index.
4220025     char        path[PATH_MAX];
4220026     int         fd_source      = -1;
4220027     int         fd_destination = -1;
4220028     char        buffer_in[BUFSIZ];
4220029     char       *buffer_out;
4220030     ssize_t     count_in;             // Read counter.
4220031     ssize_t     count_out;            // Write counter.
4220032     //
4220033     // There must be at least two arguments, plus the program name.
4220034     //
4220035     if (argc < 3)
4220036       {
4220037         usage ();
4220038         return (1);
4220039       }
4220040     //
4220041     // Select the last argument as the destination.
4220042     //
4220043     destination = argv[argc-1];
4220044     //
4220045     // Check if it is a directory and save it in a flag.
4220046     //
4220047     if (stat (destination, &file_status) == 0)
4220048       {
4220049         if (S_ISDIR (file_status.st_mode))
4220050           {
4220051             dest_is_a_dir = 1;
4220052           }
4220053       }
4220054     //
4220055     // If there are more than two arguments, verify that the last
4220056     // one is a directory.
4220057     //
4220058     if (argc > 3)
4220059       {
4220060         if (!dest_is_a_dir)
4220061           {
4220062             usage ();
4220063             fprintf (stderr, "The destination \"%s\" ",
4220064                      destination);
4220065             fprintf (stderr, "is not a directory!\n");
4220066             return (1);
4220067           }
```

```
4220068          }
4220069        //
4220070        // Scan the arguments, excluded the last, that is the destination.
4220071        //
4220072        for (a = 1; a < (argc - 1); a++)
4220073          {
4220074            //
4220075            // Source.
4220076            //
4220077            source = argv[a];
4220078            //
4220079            // Verify access permissions.
4220080            //
4220081            if (access (source, R_OK) < 0)
4220082              {
4220083                perror (source);
4220084                continue;
4220085              }
4220086            //
4220087            // Destination.
4220088            //
4220089            // If it is a directory, the destination path
4220090            // must be corrected.
4220091            //
4220092            if (dest_is_a_dir)
4220093              {
4220094                path[0] = 0;
4220095                strcat (path, destination);
4220096                strcat (path, "/");
4220097                strcat (path, basename (source));
4220098                //
4220099                // Update the destination path.
4220100                //
4220101                destination_full = path;
4220102              }
4220103            else
4220104              {
4220105                destination_full = destination;
4220106              }
4220107            //
4220108            // Check if destination file exists.
4220109            //
4220110            if (stat (destination_full, &file_status) == 0)
4220111              {
4220112                fprintf (stderr, "The destination file, \"%s\", ",
4220113                         destination_full);
4220114                fprintf (stderr, "already exists!\n");
4220115                continue;
```

```
4220116             }
4220117           //
4220118           // Everything is ready for the copy.
4220119           //
4220120           fd_source = open (source, O_RDONLY);
4220121           if (fd_source < 0)
4220122             {
4220123               perror (source);
4220124               //
4220125               // Continue with the next file.
4220126               //
4220127               continue;
4220128             }
4220129           //
4220130           fd_destination = creat (destination_full, 0777);
4220131           if (fd_destination < 0)
4220132             {
4220133               perror (destination);
4220134               close (fd_source);
4220135               //
4220136               // Continue with the next file.
4220137               //
4220138               continue;
4220139             }
4220140           //
4220141           // Copy the data.
4220142           //
4220143           while (1)
4220144             {
4220145               count_in = read (fd_source, buffer_in, (size_t) BUFSIZ);
4220146               if (count_in > 0)
4220147                 {
4220148                   for (buffer_out = buffer_in; count_in > 0;)
4220149                     {
4220150                       count_out = write (fd_destination, buffer_out,
4220151                                          (size_t) count_in);
4220152                       if (count_out < 0)
4220153                         {
4220154                           perror (destination);
4220155                           close (fd_source);
4220156                           close (fd_destination);
4220157                           return (3);
4220158                         }
4220159                       //
4220160                       // If not all data is written, continue writing,
4220161                       // but change the buffer start position and the
4220162                       // amount to be written.
4220163                       //
```

```
4220164                     buffer_out += count_out;
4220165                     count_in -= count_out;
4220166                   }
4220167               }
4220168             else if (count_in < 0)
4220169               {
4220170                 perror (source);
4220171                 close (fd_source);
4220172                 close (fd_destination);
4220173               }
4220174             else
4220175               {
4220176                 break;
4220177               }
4220178           }
4220179         //
4220180         if (close (fd_source))
4220181           {
4220182             perror (source);
4220183           }
4220184         if (close (fd_destination))
4220185           {
4220186             perror (destination);
4220187             return (4);
4220188           }
4220189       }
4220190     //
4220191     // All done.
4220192     //
4220193     return (0);
4220194 }
4220195 //----------------------------------------------------------------------
4220196 static void
4220197 usage (void)
4220198 {
4220199     fprintf (stderr, "Usage: cp OLD_NAME NEW_NAME\n");
4220200     fprintf (stderr, "       cp FILE... DIRECTORY\n");
4220201 }
```

102.1.9 applic/crt0.s

Si veda la sezione 90.9.2.

```
4230001 .extern _main
4230002 .extern __stdio_stream_setup
4230003 .extern __dirent_directory_stream_setup
4230004 .extern __atexit_setup
4230005 .extern __environment_setup
```

```
4230006 .global ___mkargv
4230007 ;-----------------------------------------------------------------------
4230008 ; Please note that, all segments are already set from the scheduler,
4230009 ; and there is also data inside the stack, so that the call to 'main()'
4230010 ; function will result as expected.
4230011 ;
4230012 ; This is a modified version of 'crt0.s' with a smaller stack size.
4230013 ;-----------------------------------------------------------------------
4230014 ; The following statement says that the code will start at "startup"
4230015 ; label.
4230016 ;-----------------------------------------------------------------------
4230017 entry startup
4230018 ;-----------------------------------------------------------------------
4230019 .text
4230020 ;-----------------------------------------------------------------------
4230021 startup:
4230022     ;
4230023     ; Jump after initial data.
4230024     ;
4230025     jmp startup_code
4230026     ;
4230027 filler:
4230028     ;
4230029     ; After four bytes, from the start, there is the
4230030     ; magic number and other data.
4230031     ;
4230032     .space (0x0004 - (filler - startup))
4230033     ;
4230034 magic:
4230035     .data4 0x6F733136   ; os16
4230036     .data4 0x6170706C   ; appl
4230037     ;
4230038 segoff:                 ;
4230039     .data2 __segoff     ; Data segment offset.
4230040 etext:                  ;
4230041     .data2 __etext      ; End of code
4230042 edata:                  ;
4230043     .data2 __edata      ; End of initialized data.
4230044 ebss:                   ;
4230045     .data2 __end        ; End of not initialized data.
4230046 stack_size:             ;
4230047     .data2 0x2000       ; Requested stack size. Every single application
4230048                         ; might change this value.
4230049     ;
4230050     ; At the next label, the work begins.
4230051     ;
4230052 .align 2
4230053 startup_code:
```

```
4230054      ;
4230055      ; Before the call to the main function, it is necessary to extract
4230056      ; the value to assign to the global variable 'environ'. It is
4230057      ; described as 'char **environ' and should contain the same address
4230058      ; pointed by 'envp'. To get this value, the stack is popped and then
4230059      ; pushed again. Please recall that the stack was prepared from
4230060      ; the process management, at the 'exec()' system call.
4230061      ;
4230062      pop ax              ; argc
4230063      pop bx              ; argv
4230064      pop cx              ; envp
4230065      mov _environ, cx    ; Variable 'environ' comes from <unistd.h>.
4230066      push cx
4230067      push bx
4230068      push ax
4230069      ;
4230070      ; Could it be enough? Of course not! To be able to handle the
4230071      ; environment, it must be copied inside the table
4230072      ; '_environment_table[][]', that is defined inside <stdlib.h>.
4230073      ; To copy the environment it is used the function
4230074      ; '_environment_setup()', passing the 'envp' pointer.
4230075      ;
4230076      push cx
4230077      call __environment_setup
4230078      add  sp, #2
4230079      ;
4230080      ; After the environment copy is done, the value for the traditional
4230081      ; variable 'environ' is updated, to point to the new array of
4230082      ; pointer. The updated value comes from variable '_environment',
4230083      ; defined inside <stdlib.h>. Then, also the 'argv' contained inside
4230084      ; the stack is replaced with the new value.
4230085      ;
4230086      mov  ax, #__environment
4230087      mov  _environ, ax
4230088      ;
4230089      pop ax              ; argc
4230090      pop bx              ; argv[][]
4230091      pop cx              ; envp[][]
4230092      mov cx, #__environment
4230093      push cx
4230094      push bx
4230095      push ax
4230096      ;
4230097      ; Setup standard I/O streams and at-exit table.
4230098      ;
4230099      call __stdio_stream_setup
4230100      call __dirent_directory_stream_setup
4230101      call __atexit_setup
```

```
4230102     ;
4230103     ; Call the main function. The arguments are already pushed inside
4230104     ; the stack.
4230105     ;
4230106     call _main
4230107     ;
4230108     ; Save the return value at the symbol 'exit_value'.
4230109     ;
4230110     mov  exit_value, ax
4230111     ;
4230112 .align 2
4230113 halt:
4230114     ;
4230115     push #2              ; Size of message.
4230116     push #exit_value     ; Pointer to the message.
4230117     push #6              ; SYS_EXIT
4230118     call _sys
4230119     add  sp, #2
4230120     add  sp, #2
4230121     add  sp, #2
4230122     ;
4230123     jmp halt
4230124     ;
4230125 ;-----------------------------------------------------------------------
4230126 .align 2
4230127 ___mkargv:       ; Symbol '___mkargv' is used by Bcc inside the function
4230128     ret          ; 'main()' and must be present for a successful
4230129                  ; compilation.
4230130 ;-----------------------------------------------------------------------
4230131 .align 2
4230132 .data
4230133 ;
4230134 exit_value:
4230135     .data2 0x0000
4230136 ;-----------------------------------------------------------------------
4230137 .align 2
4230138 .bss
```

102.1.10 applic/date.c

Si veda la sezione 92.8.

```
4240001 #include <unistd.h>
4240002 #include <stdlib.h>
4240003 #include <errno.h>
4240004 #include <time.h>
4240005 #include <ctype.h>
4240006 //----------------------------------------------------------------------
```

```
4240007 static void  usage          (void);
4240008 //----------------------------------------------------------------------
4240009 int
4240010 main (int argc, char *argv[], char *envp[])
4240011 {
4240012     struct tm *timeptr;
4240013     char       string[5];
4240014     time_t     timer;
4240015     int        length;
4240016     char      *input;
4240017     int        i;
4240018     //
4240019     // There can be at most an argument.
4240020     //
4240021     if (argc > 2)
4240022       {
4240023         usage ();
4240024         return (1);
4240025       }
4240026     //
4240027     // Check if there is no argument: must show the date.
4240028     //
4240029     if (argc == 1)
4240030       {
4240031         timer = time (NULL);
4240032         printf ("%s\n", ctime (&timer));
4240033         return (0);
4240034       }
4240035     //
4240036     // There is one argument and must be the date do set.
4240037     //
4240038     input = argv[1];
4240039     //
4240040     // First get current date, for default values.
4240041     //
4240042     timer   = time (NULL);
4240043     timeptr = gmtime (&timer);
4240044     //
4240045     // Verify to have a correct input.
4240046     //
4240047     length = (int) strlen (input);
4240048     if (length == 8 || length == 10 || length == 12)
4240049       {
4240050         for (i = 0; i < length; i++)
4240051           {
4240052             if (!isdigit (input[i]))
4240053               {
4240054                 usage ();
```

```
4240055                 return (2);
4240056               }
4240057           }
4240058         }
4240059       else
4240060         {
4240061           printf ("input: \"%s\"; length: %i\n", input, length);
4240062           usage ();
4240063           return (3);
4240064         }
4240065       //
4240066       // Select the month.
4240067       //
4240068       string[0] = input[0];
4240069       string[1] = input[1];
4240070       string[2] = '\0';
4240071       timeptr->tm_mon = atoi (string);
4240072       //
4240073       // Select the day.
4240074       //
4240075       string[0] = input[2];
4240076       string[1] = input[3];
4240077       string[2] = '\0';
4240078       timeptr->tm_mday = atoi (string);
4240079       //
4240080       // Select the hour.
4240081       //
4240082       string[0] = input[4];
4240083       string[1] = input[5];
4240084       string[2] = '\0';
4240085       timeptr->tm_hour = atoi (string);
4240086       //
4240087       // Select the minute.
4240088       //
4240089       string[0] = input[6];
4240090       string[1] = input[7];
4240091       string[2] = '\0';
4240092       timeptr->tm_min = atoi (string);
4240093       //
4240094       // Select the year: must verify if there is a century.
4240095       //
4240096       if (length == 12)
4240097         {
4240098           string[0] = input[8];
4240099           string[1] = input[9];
4240100           string[2] = input[10];
4240101           string[3] = input[11];
4240102           string[4] = '\0';
```

```
4240103          timeptr->tm_year = atoi (string);
4240104        }
4240105      else if (length == 10)
4240106        {
4240107          sprintf (string, "%04i", timeptr->tm_year);
4240108          string[2] = input[8];
4240109          string[3] = input[9];
4240110          string[4] = '\0';
4240111          timeptr->tm_year = atoi (string);
4240112        }
4240113      //
4240114      // Now convert to 'time_t'.
4240115      //
4240116      timer = mktime (timeptr);
4240117      //
4240118      // Save to the system.
4240119      //
4240120      stime (&timer);
4240121      //
4240122      return (0);
4240123  }
4240124  //----------------------------------------------------------------------
4240125  static void
4240126  usage (void)
4240127  {
4240128      fprintf (stderr, "Usage: date [MMDDHHMM[[CC]YY]]\n");
4240129  }
```

102.1.11 applic/ed.c

Si veda la sezione 92.9.

```
4250001  //----------------------------------------------------------------------
4250002  // 2009.08.18
4250003  // Modified by Daniele Giacomini for 'os16', to harmonize with it,
4250004  // even, when possible, on coding style.
4250005  //
4250006  // The original was taken form ELKS sources: 'elkscmd/misc_utils/ed.c'.
4250007  //----------------------------------------------------------------------
4250008  //
4250009  // Copyright (c) 1993 by David I. Bell
4250010  // Permission is granted to use, distribute, or modify this source,
4250011  // provided that this copyright notice remains intact.
4250012  //
4250013  // The "ed" built-in command (much simplified)
4250014  //
4250015  //----------------------------------------------------------------------
4250016
```

```
4250017 #include <stdio.h>
4250018 #include <ctype.h>
4250019 #include <unistd.h>
4250020 #include <stdbool.h>
4250021 #include <string.h>
4250022 #include <stdlib.h>
4250023 #include <fcntl.h>
4250024 //----------------------------------------------------------------------
4250025 #define isoctal(ch)   (((ch) >= '0') && ((ch) <= '7'))
4250026 #define USERSIZE      1024        /* max line length typed in by user */
4250027 #define INITBUFSIZE   1024        /* initial buffer size */
4250028 //----------------------------------------------------------------------
4250029 typedef int num_t;
4250030 typedef int len_t;
4250031 //
4250032 // The following is the type definition of structure `line_t', but the
4250033 // structure contains pointers to the same kind of type. With the
4250034 // compiler Bcc, it is the only way to declare it.
4250035 //
4250036 typedef struct line line_t;
4250037 //
4250038 struct line {
4250039     line_t *next;
4250040     line_t *prev;
4250041     len_t   len;
4250042     char    data[1];
4250043 };
4250044 //
4250045 static  line_t  lines;
4250046 static  line_t *curline;
4250047 static  num_t   curnum;
4250048 static  num_t   lastnum;
4250049 static  num_t   marks[26];
4250050 static  bool    dirty;
4250051 static  char    *filename;
4250052 static  char    searchstring[USERSIZE];
4250053 //
4250054 static  char   *bufbase;
4250055 static  char   *bufptr;
4250056 static  len_t   bufused;
4250057 static  len_t   bufsize;
4250058 //----------------------------------------------------------------------
4250059 static  void    docommands  (void);
4250060 static  void    subcommand  (char *cp, num_t num1, num_t num2);
4250061 static  bool    getnum      (char **retcp, bool *rethavenum,
4250062                              num_t *retnum);
4250063 static  bool    setcurnum   (num_t num);
4250064 static  bool    initedit    (void);
```

```
4250065 static  void    termedit    (void);
4250066 static  void    addlines    (num_t num);
4250067 static  bool    insertline  (num_t num, char *data, len_t len);
4250068 static  bool    deletelines (num_t num1, num_t num2);
4250069 static  bool    printlines  (num_t num1, num_t num2, bool expandflag);
4250070 static  bool    writelines  (char *file, num_t num1, num_t num2);
4250071 static  bool    readlines   (char *file, num_t num);
4250072 static  num_t   searchlines (char *str, num_t num1, num_t num2);
4250073 static  len_t   findstring  (line_t *lp, char *str, len_t len,
4250074                               len_t offset);
4250075 static  line_t *findline    (num_t num);
4250076 //----------------------------------------------------------------------
4250077 // Main.
4250078 //----------------------------------------------------------------------
4250079 int
4250080 main (int argc, char *argv[], char *envp[])
4250081 {
4250082     if (!initedit ()) return (2);
4250083     //
4250084     if (argc > 1)
4250085       {
4250086         filename = strdup (argv[1]);
4250087         if (filename == NULL)
4250088           {
4250089             fprintf (stderr, "No memory\n");
4250090             termedit ();
4250091             return (1);
4250092           }
4250093         //
4250094         if (!readlines (filename, 1))
4250095           {
4250096             termedit ();
4250097             return (0);
4250098           }
4250099         //
4250100         if (lastnum) setcurnum(1);
4250101         //
4250102         dirty = false;
4250103       }
4250104     //
4250105     docommands ();
4250106     //
4250107     termedit ();
4250108     return (0);
4250109 }
4250110 //----------------------------------------------------------------------
4250111 // Read commands until we are told to stop.
4250112 //----------------------------------------------------------------------
```

```
4250113 void
4250114 docommands (void)
4250115 {
4250116     char   *cp;
4250117     int     len;
4250118     num_t   num1;
4250119     num_t   num2;
4250120     bool    have1;
4250121     bool    have2;
4250122     char    buf[USERSIZE];
4250123     //
4250124     while (true)
4250125       {
4250126         printf(": ");
4250127         fflush (stdout);
4250128         //
4250129         if (fgets (buf, sizeof(buf), stdin) == NULL)
4250130           {
4250131             return;
4250132           }
4250133         //
4250134         len = strlen (buf);
4250135         if (len == 0)
4250136           {
4250137             return;
4250138           }
4250139         //
4250140         cp = &buf[len - 1];
4250141         if (*cp != '\n')
4250142           {
4250143             fprintf(stderr, "Command line too long\n");
4250144             do
4250145               {
4250146                 len = fgetc(stdin);
4250147               }
4250148             while ((len != EOF) && (len != '\n'));
4250149             //
4250150             continue;
4250151           }
4250152         //
4250153         while ((cp > buf) && isblank (cp[-1]))
4250154           {
4250155             cp--;
4250156           }
4250157         //
4250158         *cp = '\0';
4250159         //
4250160         cp = buf;
```

```
4250161             //
4250162             while (isblank (*cp))
4250163               {
4250164                 //*cp++;
4250165                 cp++;
4250166               }
4250167             //
4250168             have1 = false;
4250169             have2 = false;
4250170             //
4250171             if ((curnum == 0) && (lastnum > 0))
4250172               {
4250173                 curnum = 1;
4250174                 curline = lines.next;
4250175               }
4250176             //
4250177             if (!getnum (&cp, &have1, &num1))
4250178               {
4250179                 continue;
4250180               }
4250181             //
4250182             while (isblank (*cp))
4250183               {
4250184                 cp++;
4250185               }
4250186             //
4250187             if (*cp == ',')
4250188               {
4250189                 cp++;
4250190                 if (!getnum (&cp, &have2, &num2))
4250191                   {
4250192                     continue;
4250193                   }
4250194                 //
4250195                 if (!have1)
4250196                   {
4250197                     num1 = 1;
4250198                   }
4250199                 if (!have2)
4250200                   {
4250201                     num2 = lastnum;
4250202                   }
4250203                 have1 = true;
4250204                 have2 = true;
4250205               }
4250206             //
4250207             if (!have1)
4250208               {
```

```
4250209                 num1 = curnum;
4250210               }
4250211             if (!have2)
4250212               {
4250213                 num2 = num1;
4250214               }
4250215             //
4250216             // Command interpretation switch.
4250217             //
4250218             switch (*cp++)
4250219               {
4250220                 case 'a':
4250221                     addlines (num1 + 1);
4250222                     break;
4250223                     //
4250224                 case 'c':
4250225                     deletelines (num1, num2);
4250226                     addlines (num1);
4250227                     break;
4250228                     //
4250229                 case 'd':
4250230                     deletelines (num1, num2);
4250231                     break;
4250232                     //
4250233                 case 'f':
4250234                     if (*cp && !isblank (*cp))
4250235                       {
4250236                         fprintf (stderr, "Bad file command\n");
4250237                         break;
4250238                       }
4250239                     //
4250240                     while (isblank (*cp))
4250241                       {
4250242                         cp++;
4250243                       }
4250244                     if (*cp == '\0')
4250245                       {
4250246                         if (filename)
4250247                           {
4250248                             printf ("\"%s\"\n", filename);
4250249                           }
4250250                         else
4250251                           {
4250252                             printf ("No filename\n");
4250253                           }
4250254                         break;
4250255                       }
4250256                     //
```

```
4250257                     cp = strdup (cp);
4250258                     //
4250259                     if (cp == NULL)
4250260                       {
4250261                         fprintf (stderr, "No memory for filename\n");
4250262                         break;
4250263                       }
4250264                     //
4250265                     if (filename)
4250266                       {
4250267                         free(filename);
4250268                       }
4250269                     //
4250270                     filename = cp;
4250271                     break;
4250272                     //
4250273                 case 'i':
4250274                     addlines (num1);
4250275                     break;
4250276                     //
4250277                 case 'k':
4250278                     while (isblank(*cp))
4250279                       {
4250280                         cp++;
4250281                       }
4250282                     //
4250283                     if ((*cp < 'a') || (*cp > 'a') || cp[1])
4250284                       {
4250285                         fprintf (stderr, "Bad mark name\n");
4250286                         break;
4250287                       }
4250288                     //
4250289                     marks[*cp - 'a'] = num2;
4250290                     break;
4250291                     //
4250292                 case 'l':
4250293                     printlines (num1, num2, true);
4250294                     break;
4250295                     //
4250296                 case 'p':
4250297                     printlines (num1, num2, false);
4250298                     break;
4250299                     //
4250300                 case 'q':
4250301                     while (isblank(*cp))
4250302                       {
4250303                         cp++;
4250304                       }
```

```
4250305                    //
4250306                    if (have1 || *cp)
4250307                      {
4250308                        fprintf (stderr, "Bad quit command\n");
4250309                        break;
4250310                      }
4250311                    //
4250312                    if (!dirty)
4250313                      {
4250314                        return;
4250315                      }
4250316                    //
4250317                    printf ("Really quit? ");
4250318                    fflush (stdout);
4250319                    //
4250320                    buf[0] = '\0';
4250321                    fgets (buf, sizeof(buf), stdin);
4250322                    cp = buf;
4250323                    //
4250324                    while (isblank (*cp))
4250325                      {
4250326                        cp++;
4250327                      }
4250328                    //
4250329                    if ((*cp == 'y') || (*cp == 'Y'))
4250330                      {
4250331                        return;
4250332                      }
4250333                    //
4250334                    break;
4250335                    //
4250336                case 'r':
4250337                    if (*cp && !isblank(*cp))
4250338                      {
4250339                        fprintf (stderr, "Bad read command\n");
4250340                        break;
4250341                      }
4250342                    //
4250343                    while (isblank(*cp))
4250344                      {
4250345                        cp++;
4250346                      }
4250347                    //
4250348                    if (*cp == '\0')
4250349                      {
4250350                        fprintf (stderr, "No filename\n");
4250351                        break;
4250352                      }
```

```
4250353                 //
4250354                 if (!have1)
4250355                   {
4250356                     num1 = lastnum;
4250357                   }
4250358                 //
4250359                 // Open the file and add to the buffer
4250360                 // at the next line.
4250361                 //
4250362                 if (readlines (cp, num1 + 1))
4250363                   {
4250364                     //
4250365                     // If the file open fails, just
4250366                     // break the command.
4250367                     //
4250368                     break;
4250369                   }
4250370                 //
4250371                 // Set the default file name, if no
4250372                 // previous name is available.
4250373                 //
4250374                 if (filename == NULL)
4250375                   {
4250376                     filename = strdup (cp);
4250377                   }
4250378                 //
4250379                 break;
4250380
4250381             case 's':
4250382                 subcommand (cp, num1, num2);
4250383                 break;
4250384                 //
4250385             case 'w':
4250386                 if (*cp && !isblank(*cp))
4250387                   {
4250388                     fprintf(stderr, "Bad write command\n");
4250389                     break;
4250390                   }
4250391                 //
4250392                 while (isblank(*cp))
4250393                   {
4250394                     cp++;
4250395                   }
4250396                 //
4250397                 if (!have1)
4250398                   {
4250399                     num1 = 1;
4250400                     num2 = lastnum;
```

```
4250401                         }
4250402                     //
4250403                     // If the file name is not specified, use the
4250404                     // default one.
4250405                     //
4250406                     if (*cp == '\0')
4250407                       {
4250408                         cp = filename;
4250409                       }
4250410                     //
4250411                     // If even the default file name is not specified,
4250412                     // tell it.
4250413                     //
4250414                     if (cp == NULL)
4250415                       {
4250416                         fprintf (stderr, "No file name specified\n");
4250417                         break;
4250418                       }
4250419                     //
4250420                     // Write the file.
4250421                     //
4250422                     writelines (cp, num1, num2);
4250423                     //
4250424                     break;
4250425                     //
4250426                 case 'z':
4250427                     switch (*cp)
4250428                       {
4250429                         case '-':
4250430                             printlines (curnum-21, curnum, false);
4250431                             break;
4250432                         case '.':
4250433                             printlines (curnum-11, curnum+10, false);
4250434                             break;
4250435                         default:
4250436                             printlines (curnum, curnum+21, false);
4250437                             break;
4250438                       }
4250439                     break;
4250440                     //
4250441                 case '.':
4250442                     if (have1)
4250443                       {
4250444                         fprintf (stderr, "No arguments allowed\n");
4250445                         break;
4250446                       }
4250447                     printlines (curnum, curnum, false);
4250448                     break;
```

```
4250449                 //
4250450             case '-':
4250451                 if (setcurnum (curnum - 1))
4250452                   {
4250453                     printlines (curnum, curnum, false);
4250454                   }
4250455                 break;
4250456                 //
4250457             case '=':
4250458                 printf ("%d\n", num1);
4250459                 break;
4250460                 //
4250461             case '\0':
4250462                 if (have1)
4250463                   {
4250464                     printlines (num2, num2, false);
4250465                     break;
4250466                   }
4250467                 //
4250468                 if (setcurnum (curnum + 1))
4250469                   {
4250470                     printlines (curnum, curnum, false);
4250471                   }
4250472                 break;
4250473                 //
4250474             default:
4250475                 fprintf (stderr, "Unimplemented command\n");
4250476                 break;
4250477           }
4250478       }
4250479 }
4250480 //------------------------------------------------------------------------
4250481 // Do the substitute command.
4250482 // The current line is set to the last substitution done.
4250483 //------------------------------------------------------------------------
4250484 void
4250485 subcommand (char *cp, num_t num1, num_t num2)
4250486 {
4250487     int     delim;
4250488     char   *oldstr;
4250489     char   *newstr;
4250490     len_t   oldlen;
4250491     len_t   newlen;
4250492     len_t   deltalen;
4250493     len_t   offset;
4250494     line_t *lp;
4250495     line_t *nlp;
4250496     bool    globalflag;
```

```
4250497     bool    printflag;
4250498     bool    didsub;
4250499     bool    needprint;
4250500 
4250501     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4250502       {
4250503         fprintf (stderr, "Bad line range for substitute\n");
4250504         return;
4250505       }
4250506     //
4250507     globalflag = false;
4250508     printflag = false;
4250509     didsub = false;
4250510     needprint = false;
4250511     //
4250512     if (isblank (*cp) || (*cp == '\0'))
4250513       {
4250514         fprintf (stderr, "Bad delimiter for substitute\n");
4250515         return;
4250516       }
4250517     //
4250518     delim = *cp++;
4250519     oldstr = cp;
4250520     //
4250521     cp = strchr (cp, delim);
4250522     //
4250523     if (cp == NULL)
4250524       {
4250525         fprintf (stderr, "Missing 2nd delimiter for substitute\n");
4250526         return;
4250527       }
4250528     //
4250529     *cp++ = '\0';
4250530     //
4250531     newstr = cp;
4250532     cp = strchr (cp, delim);
4250533     //
4250534     if (cp)
4250535       {
4250536         *cp++ = '\0';
4250537       }
4250538     else
4250539       {
4250540         cp = "";
4250541       }
4250542     while (*cp)
4250543       {
4250544         switch (*cp++)
```

```
4250545             {
4250546               case 'g':
4250547                   globalflag = true;
4250548                   break;
4250549                   //
4250550               case 'p':
4250551                   printflag = true;
4250552                   break;
4250553                   //
4250554               default:
4250555                   fprintf (stderr, "Unknown option for substitute\n");
4250556                   return;
4250557             }
4250558         }
4250559       //
4250560       if (*oldstr == '\0')
4250561         {
4250562           if (searchstring[0] == '\0')
4250563             {
4250564               fprintf (stderr, "No previous search string\n");
4250565               return;
4250566             }
4250567           oldstr = searchstring;
4250568         }
4250569       //
4250570       if (oldstr != searchstring)
4250571         {
4250572           strcpy (searchstring, oldstr);
4250573         }
4250574       //
4250575       lp = findline (num1);
4250576       if (lp == NULL)
4250577         {
4250578           return;
4250579         }
4250580       //
4250581       oldlen = strlen(oldstr);
4250582       newlen = strlen(newstr);
4250583       deltalen = newlen - oldlen;
4250584       offset = 0;
4250585       //
4250586       while (num1 <= num2)
4250587         {
4250588           offset = findstring (lp, oldstr, oldlen, offset);
4250589           if (offset < 0)
4250590             {
4250591               if (needprint)
4250592                 {
```

```
4250593                     printlines (num1, num1, false);
4250594                     needprint = false;
4250595                   }
4250596                 //
4250597                 offset = 0;
4250598                 lp = lp->next;
4250599                 num1++;
4250600                 continue;
4250601               }
4250602             //
4250603             needprint = printflag;
4250604             didsub = true;
4250605             dirty = true;
4250606
4250607             //-----------------------------------------------------------
4250608             // If the replacement string is the same size or shorter
4250609             // than the old string, then the substitution is easy.
4250610             //-----------------------------------------------------------
4250611
4250612             if (deltalen <= 0)
4250613               {
4250614                 memcpy (&lp->data[offset], newstr, newlen);
4250615                 //
4250616                 if (deltalen)
4250617                   {
4250618                     memcpy (&lp->data[offset + newlen],
4250619                             &lp->data[offset + oldlen],
4250620                             lp->len - offset - oldlen);
4250621                     //
4250622                     lp->len += deltalen;
4250623                   }
4250624                 //
4250625                 offset += newlen;
4250626                 //
4250627                 if (globalflag)
4250628                   {
4250629                     continue;
4250630                   }
4250631                 //
4250632                 if (needprint)
4250633                   {
4250634                     printlines(num1, num1, false);
4250635                     needprint = false;
4250636                   }
4250637                 //
4250638                 lp = nlp->next;
4250639                 num1++;
4250640                 continue;
```

```
4250641               }
4250642
4250643             //------------------------------------------------------------
4250644             // The new string is larger, so allocate a new line
4250645             // structure and use that.  Link it in in place of
4250646             // the old line structure.
4250647             //------------------------------------------------------------
4250648
4250649             nlp = (line_t *) malloc (sizeof (line_t) + lp->len + deltalen);
4250650             //
4250651             if (nlp == NULL)
4250652               {
4250653                 fprintf (stderr, "Cannot get memory for line\n");
4250654                 return;
4250655               }
4250656             //
4250657             nlp->len = lp->len + deltalen;
4250658             //
4250659             memcpy (nlp->data, lp->data, offset);
4250660             //
4250661             memcpy (&nlp->data[offset], newstr, newlen);
4250662             //
4250663             memcpy (&nlp->data[offset + newlen],
4250664                     &lp->data[offset + oldlen],
4250665                     lp->len - offset - oldlen);
4250666             //
4250667             nlp->next = lp->next;
4250668             nlp->prev = lp->prev;
4250669             nlp->prev->next = nlp;
4250670             nlp->next->prev = nlp;
4250671             //
4250672             if (curline == lp)
4250673               {
4250674                 curline = nlp;
4250675               }
4250676             //
4250677             free(lp);
4250678             lp = nlp;
4250679             //
4250680             offset += newlen;
4250681             //
4250682             if (globalflag)
4250683               {
4250684                 continue;
4250685               }
4250686             //
4250687             if (needprint)
4250688               {
```

```
4250689                 printlines (num1, num1, false);
4250690                 needprint = false;
4250691               }
4250692             //
4250693             lp = lp->next;
4250694             num1++;
4250695           }
4250696         //
4250697         if (!didsub)
4250698           {
4250699             fprintf (stderr, "No substitutions found for \"%s\"\n", oldstr);
4250700           }
4250701 }
4250702 //----------------------------------------------------------------------
4250703 // Search a line for the specified string starting at the specified
4250704 // offset in the line.  Returns the offset of the found string, or -1.
4250705 //----------------------------------------------------------------------
4250706 len_t
4250707 findstring (line_t *lp, char *str, len_t len, len_t offset)
4250708 {
4250709         len_t    left;
4250710         char    *cp;
4250711         char    *ncp;
4250712         //
4250713         cp = &lp->data[offset];
4250714         left = lp->len - offset;
4250715         //
4250716         while (left >= len)
4250717           {
4250718             ncp = memchr(cp, *str, left);
4250719             if (ncp == NULL)
4250720               {
4250721                 return (len_t) -1;
4250722               }
4250723             //
4250724             left -= (ncp - cp);
4250725             if (left < len)
4250726               {
4250727                 return (len_t) -1;
4250728               }
4250729             //
4250730             cp = ncp;
4250731             if (memcmp(cp, str, len) == 0)
4250732               {
4250733                 return (len_t) (cp - lp->data);
4250734               }
4250735             //
4250736             cp++;
```

```
4250737             left--;
4250738           }
4250739         //
4250740         return (len_t) -1;
4250741 }
4250742 //----------------------------------------------------------------------
4250743 // Add lines which are typed in by the user.
4250744 // The lines are inserted just before the specified line number.
4250745 // The lines are terminated by a line containing a single dot (ugly!),
4250746 // or by an end of file.
4250747 //----------------------------------------------------------------------
4250748 void
4250749 addlines (num_t num)
4250750 {
4250751     int     len;
4250752     char    buf[USERSIZE + 1];
4250753     //
4250754     while (fgets (buf, sizeof (buf), stdin))
4250755       {
4250756         if ((buf[0] == '.') && (buf[1] == '\n') && (buf[2] == '\0'))
4250757           {
4250758             return;
4250759           }
4250760         //
4250761         len = strlen (buf);
4250762         //
4250763         if (len == 0)
4250764           {
4250765             return;
4250766           }
4250767         //
4250768         if (buf[len - 1] != '\n')
4250769          {
4250770             fprintf (stderr, "Line too long\n");
4250771             //
4250772             do
4250773               {
4250774                 len = fgetc(stdin);
4250775               }
4250776             while ((len != EOF) && (len != '\n'));
4250777             //
4250778             return;
4250779           }
4250780         //
4250781         if (!insertline (num++, buf, len))
4250782           {
4250783             return;
4250784           }
```

```
4250785         }
4250786 }
4250787 //----------------------------------------------------------------------
4250788 // Parse a line number argument if it is present.  This is a sum
4250789 // or difference of numbers, '.', '$', 'x, or a search string.
4250790 // Returns true if successful (whether or not there was a number).
4250791 // Returns false if there was a parsing error, with a message output.
4250792 // Whether there was a number is returned indirectly, as is the number.
4250793 // The character pointer which stopped the scan is also returned.
4250794 //----------------------------------------------------------------------
4250795 static bool
4250796 getnum (char **retcp, bool *rethavenum, num_t *retnum)
4250797 {
4250798     char    *cp;
4250799     char    *str;
4250800     bool     havenum;
4250801     num_t    value;
4250802     num_t    num;
4250803     num_t    sign;
4250804     //
4250805     cp = *retcp;
4250806     havenum = false;
4250807     value = 0;
4250808     sign = 1;
4250809     //
4250810     while (true)
4250811       {
4250812         while (isblank(*cp))
4250813           {
4250814             cp++;
4250815           }
4250816         //
4250817         switch (*cp)
4250818           {
4250819             case '.':
4250820                 havenum = true;
4250821                 num = curnum;
4250822                 cp++;
4250823                 break;
4250824                 //
4250825             case '$':
4250826                 havenum = true;
4250827                 num = lastnum;
4250828                 cp++;
4250829                 break;
4250830                 //
4250831             case '\'':
4250832                 cp++;
```

```
4250833                 if ((*cp < 'a') || (*cp > 'z'))
4250834                   {
4250835                     fprintf (stderr, "Bad mark name\n");
4250836                     return false;
4250837                   }
4250838                 //
4250839                 havenum = true;
4250840                 num = marks[*cp++ - 'a'];
4250841                 break;
4250842                 //
4250843             case '/':
4250844                 str = ++cp;
4250845                 cp = strchr (str, '/');
4250846                 if (cp)
4250847                   {
4250848                     *cp++ = '\0';
4250849                   }
4250850                 else
4250851                   {
4250852                     cp = "";
4250853                   }
4250854                 num = searchlines (str, curnum, lastnum);
4250855                 if (num == 0)
4250856                   {
4250857                     return false;
4250858                   }
4250859                 //
4250860                 havenum = true;
4250861                 break;
4250862                 //
4250863             default:
4250864                 if (!isdigit (*cp))
4250865                  {
4250866                     *retcp = cp;
4250867                     *rethavenum = havenum;
4250868                     *retnum = value;
4250869                     return true;
4250870                   }
4250871                 //
4250872                 num = 0;
4250873                 while (isdigit(*cp))
4250874                   {
4250875                     num = num * 10 + *cp++ - '0';
4250876                   }
4250877                 havenum = true;
4250878                 break;
4250879           }
4250880         //
```

```
4250881         value += num * sign;
4250882         //
4250883         while (isblank (*cp))
4250884           {
4250885             cp++;
4250886           }
4250887         //
4250888         switch (*cp)
4250889           {
4250890             case '-':
4250891                 sign = -1;
4250892                 cp++;
4250893                 break;
4250894                 //
4250895             case '+':
4250896                 sign = 1;
4250897                 cp++;
4250898                 break;
4250899                 //
4250900             default:
4250901                 *retcp = cp;
4250902                 *rethavenum = havenum;
4250903                 *retnum = value;
4250904                 return true;
4250905           }
4250906     }
4250907 }
4250908 //----------------------------------------------------------------------
4250909 // Initialize everything for editing.
4250910 //----------------------------------------------------------------------
4250911 bool
4250912 initedit (void)
4250913 {
4250914     int i;
4250915     //
4250916     bufsize = INITBUFSIZE;
4250917     bufbase = malloc (bufsize);
4250918     //
4250919     if (bufbase == NULL)
4250920       {
4250921         fprintf (stderr, "No memory for buffer\n");
4250922         return false;
4250923       }
4250924     //
4250925     bufptr = bufbase;
4250926     bufused = 0;
4250927     //
4250928     lines.next = &lines;
```

```
4250929     lines.prev = &lines;
4250930     //
4250931     curline = NULL;
4250932     curnum = 0;
4250933     lastnum = 0;
4250934     dirty = false;
4250935     filename = NULL;
4250936     searchstring[0] = '\0';
4250937     //
4250938     for (i = 0; i < 26; i++)
4250939       {
4250940         marks[i] = 0;
4250941       }
4250942     //
4250943     return true;
4250944 }
4250945 //----------------------------------------------------------------------
4250946 // Finish editing.
4250947 //----------------------------------------------------------------------
4250948 void
4250949 termedit (void)
4250950 {
4250951     if (bufbase) free(bufbase);
4250952     bufbase = NULL;
4250953     //
4250954     bufptr = NULL;
4250955     bufsize = 0;
4250956     bufused = 0;
4250957     //
4250958     if (filename) free(filename);
4250959     filename = NULL;
4250960     //
4250961     searchstring[0] = '\0';
4250962     //
4250963     if (lastnum) deletelines (1, lastnum);
4250964     //
4250965     lastnum = 0;
4250966     curnum = 0;
4250967     curline = NULL;
4250968 }
4250969 //----------------------------------------------------------------------
4250970 // Read lines from a file at the specified line number.
4250971 // Returns true if the file was successfully read.
4250972 //----------------------------------------------------------------------
4250973 bool
4250974 readlines (char *file, num_t num)
4250975 {
4250976     int     fd;
```

```
4250977     int      cc;
4250978     len_t    len;
4250979     len_t    linecount;
4250980     len_t    charcount;
4250981     char    *cp;
4250982     //
4250983     if ((num < 1) || (num > lastnum + 1))
4250984       {
4250985         fprintf (stderr, "Bad line for read\n");
4250986         return false;
4250987       }
4250988     //
4250989     fd = open (file, O_RDONLY);
4250990     if (fd < 0)
4250991       {
4250992         perror (file);
4250993         return false;
4250994       }
4250995     //
4250996     bufptr = bufbase;
4250997     bufused = 0;
4250998     linecount = 0;
4250999     charcount = 0;
4251000     //
4251001     printf ("\"%s\", ", file);
4251002     fflush(stdout);
4251003     //
4251004     do
4251005       {
4251006         cp = memchr(bufptr, '\n', bufused);
4251007         if (cp)
4251008           {
4251009             len = (cp - bufptr) + 1;
4251010             //
4251011             if (!insertline (num, bufptr, len))
4251012               {
4251013                 close (fd);
4251014                 return false;
4251015               }
4251016             //
4251017             bufptr += len;
4251018             bufused -= len;
4251019             charcount += len;
4251020             linecount++;
4251021             num++;
4251022             continue;
4251023           }
4251024         //
```

```
4251025            if (bufptr != bufbase)
4251026              {
4251027                memcpy (bufbase, bufptr, bufused);
4251028                bufptr = bufbase + bufused;
4251029              }
4251030            //
4251031            if (bufused >= bufsize)
4251032              {
4251033                len = (bufsize * 3) / 2;
4251034                cp = realloc (bufbase, len);
4251035                if (cp == NULL)
4251036                  {
4251037                    fprintf (stderr, "No memory for buffer\n");
4251038                    close (fd);
4251039                    return false;
4251040                  }
4251041                //
4251042                bufbase = cp;
4251043                bufptr = bufbase + bufused;
4251044                bufsize = len;
4251045              }
4251046            //
4251047            cc = read (fd, bufptr, bufsize - bufused);
4251048            bufused += cc;
4251049            bufptr = bufbase;
4251050          }
4251051        while (cc > 0);
4251052        //
4251053        if (cc < 0)
4251054          {
4251055            perror (file);
4251056            close (fd);
4251057            return false;
4251058          }
4251059        //
4251060        if (bufused)
4251061          {
4251062            if (!insertline (num, bufptr, bufused))
4251063              {
4251064                close (fd);
4251065                return -1;
4251066              }
4251067            linecount++;
4251068            charcount += bufused;
4251069          }
4251070        //
4251071        close (fd);
4251072        //
```

```
4251073     printf ("%d lines%s, %d chars\n",
4251074             linecount,
4251075             (bufused ? " (incomplete)" : ""),
4251076             charcount);
4251077     //
4251078     return true;
4251079 }
4251080 //----------------------------------------------------------------------
4251081 // Write the specified lines out to the specified file.
4251082 // Returns true if successful, or false on an error with a message
4251083 // output.
4251084 //----------------------------------------------------------------------
4251085 bool
4251086 writelines (char *file, num_t num1, num_t num2)
4251087 {
4251088     int     fd;
4251089     line_t *lp;
4251090     len_t   linecount;
4251091     len_t   charcount;
4251092     //
4251093     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251094       {
4251095         fprintf (stderr, "Bad line range for write\n");
4251096         return false;
4251097       }
4251098     //
4251099     linecount = 0;
4251100     charcount = 0;
4251101     //
4251102     fd = creat (file, 0666);
4251103     if (fd < 0)
4251104       {
4251105         perror (file);
4251106         return false;
4251107       }
4251108     //
4251109     printf("\"%s\", ", file);
4251110     fflush (stdout);
4251111     //
4251112     lp = findline (num1);
4251113     if (lp == NULL)
4251114       {
4251115         close (fd);
4251116         return false;
4251117       }
4251118     //
4251119     while (num1++ <= num2)
4251120       {
```

```
4251121          if (write(fd, lp->data, lp->len) != lp->len)
4251122            {
4251123              perror(file);
4251124              close(fd);
4251125              return false;
4251126            }
4251127          //
4251128          charcount += lp->len;
4251129          linecount++;
4251130          lp = lp->next;
4251131        }
4251132      //
4251133      if (close(fd) < 0)
4251134        {
4251135          perror(file);
4251136          return false;
4251137        }
4251138      //
4251139      printf ("%d lines, %d chars\n", linecount, charcount);
4251140      //
4251141      return true;
4251142  }
4251143  //----------------------------------------------------------------------
4251144  // Print lines in a specified range.
4251145  // The last line printed becomes the current line.
4251146  // If expandflag is true, then the line is printed specially to
4251147  // show magic characters.
4251148  //----------------------------------------------------------------------
4251149  bool
4251150  printlines (num_t num1, num_t num2, bool expandflag)
4251151  {
4251152      line_t        *lp;
4251153      unsigned char *cp;
4251154      int            ch;
4251155      len_t          count;
4251156      //
4251157      if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251158        {
4251159          fprintf (stderr, "Bad line range for print\n");
4251160          return false;
4251161        }
4251162      //
4251163      lp = findline (num1);
4251164      if (lp == NULL)
4251165        {
4251166          return false;
4251167        }
4251168      //
```

```
4251169     while (num1 <= num2)
4251170       {
4251171         if (!expandflag)
4251172           {
4251173             write (STDOUT_FILENO, lp->data, lp->len);
4251174             setcurnum (num1++);
4251175             lp = lp->next;
4251176             continue;
4251177           }
4251178 
4251179         //-----------------------------------------------------------
4251180         // Show control characters and characters with the
4251181         // high bit set specially.
4251182         //-----------------------------------------------------------
4251183 
4251184         cp = (unsigned char *) lp->data;
4251185         count = lp->len;
4251186         //
4251187         if ((count > 0) && (cp[count - 1] == '\n'))
4251188           {
4251189             count--;
4251190           }
4251191         //
4251192         while (count-- > 0)
4251193           {
4251194             ch = *cp++;
4251195             if (ch & 0x80)
4251196               {
4251197                 fputs ("M-", stdout);
4251198                 ch &= 0x7f;
4251199               }
4251200             if (ch < ' ')
4251201               {
4251202                 fputc ('^', stdout);
4251203                 ch += '@';
4251204               }
4251205             if (ch == 0x7f)
4251206               {
4251207                 fputc ('^', stdout);
4251208                 ch = '?';
4251209               }
4251210             fputc (ch, stdout);
4251211           }
4251212         //
4251213         fputs ("$\n", stdout);
4251214         //
4251215         setcurnum (num1++);
4251216         lp = lp->next;
```

```
4251217          }
4251218        //
4251219        return true;
4251220    }
4251221    //----------------------------------------------------------------------
4251222    // Insert a new line with the specified text.
4251223    // The line is inserted so as to become the specified line,
4251224    // thus pushing any existing and further lines down one.
4251225    // The inserted line is also set to become the current line.
4251226    // Returns true if successful.
4251227    //----------------------------------------------------------------------
4251228    bool
4251229    insertline (num_t num, char *data, len_t len)
4251230    {
4251231        line_t    *newlp;
4251232        line_t    *lp;
4251233        //
4251234        if ((num < 1) || (num > lastnum + 1))
4251235          {
4251236            fprintf (stderr, "Inserting at bad line number\n");
4251237            return false;
4251238          }
4251239        //
4251240        newlp = (line_t *) malloc (sizeof (line_t) + len - 1);
4251241        if (newlp == NULL)
4251242          {
4251243            fprintf (stderr, "Failed to allocate memory for line\n");
4251244            return false;
4251245          }
4251246        //
4251247        memcpy (newlp->data, data, len);
4251248        newlp->len = len;
4251249        //
4251250        if (num > lastnum)
4251251          {
4251252            lp = &lines;
4251253          }
4251254        else
4251255          {
4251256            lp = findline (num);
4251257            if (lp == NULL)
4251258              {
4251259                free ((char *) newlp);
4251260                return false;
4251261              }
4251262          }
4251263        //
4251264        newlp->next = lp;
```

```
4251265     newlp->prev = lp->prev;
4251266     lp->prev->next = newlp;
4251267     lp->prev = newlp;
4251268     //
4251269     lastnum++;
4251270     dirty = true;
4251271     //
4251272     return setcurnum (num);
4251273 }
4251274 //----------------------------------------------------------------------
4251275 // Delete lines from the given range.
4251276 //----------------------------------------------------------------------
4251277 bool
4251278 deletelines (num_t num1, num_t num2)
4251279 {
4251280     line_t   *lp;
4251281     line_t   *nlp;
4251282     line_t   *plp;
4251283     num_t     count;
4251284     //
4251285     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251286       {
4251287         fprintf (stderr, "Bad line numbers for delete\n");
4251288         return false;
4251289       }
4251290     //
4251291     lp = findline (num1);
4251292     if (lp == NULL)
4251293       {
4251294         return false;
4251295       }
4251296     //
4251297     if ((curnum >= num1) && (curnum <= num2))
4251298       {
4251299         if (num2 < lastnum)
4251300           {
4251301             setcurnum (num2 + 1);
4251302           }
4251303         else if (num1 > 1)
4251304           {
4251305             setcurnum (num1 - 1);
4251306           }
4251307         else
4251308           {
4251309             curnum = 0;
4251310           }
4251311       }
4251312     //
```

```
4251313     count = num2 - num1 + 1;
4251314     //
4251315     if (curnum > num2)
4251316       {
4251317         curnum -= count;
4251318       }
4251319     //
4251320     lastnum -= count;
4251321     //
4251322     while (count-- > 0)
4251323       {
4251324         nlp = lp->next;
4251325         plp = lp->prev;
4251326         plp->next = nlp;
4251327         nlp->prev = plp;
4251328         lp->next = NULL;
4251329         lp->prev = NULL;
4251330         lp->len = 0;
4251331         free(lp);
4251332         lp = nlp;
4251333       }
4251334     //
4251335     dirty = true;
4251336     //
4251337     return true;
4251338 }
4251339 //----------------------------------------------------------------------
4251340 // Search for a line which contains the specified string.
4251341 // If the string is NULL, then the previously searched for string
4251342 // is used.  The currently searched for string is saved for future use.
4251343 // Returns the line number which matches, or 0 if there was no match
4251344 // with an error printed.
4251345 //----------------------------------------------------------------------
4251346 num_t
4251347 searchlines (char *str, num_t num1, num_t num2)
4251348 {
4251349     line_t *lp;
4251350     int     len;
4251351     //
4251352     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251353       {
4251354         fprintf (stderr, "Bad line numbers for search\n");
4251355         return 0;
4251356       }
4251357     //
4251358     if (*str == '\0')
4251359       {
4251360         if (searchstring[0] == '\0')
```

```
4251361             {
4251362               fprintf(stderr, "No previous search string\n");
4251363               return 0;
4251364             }
4251365           str = searchstring;
4251366         }
4251367       //
4251368       if (str != searchstring)
4251369         {
4251370           strcpy(searchstring, str);
4251371         }
4251372       //
4251373       len = strlen(str);
4251374       //
4251375       lp = findline (num1);
4251376       if (lp == NULL)
4251377         {
4251378           return 0;
4251379         }
4251380       //
4251381       while (num1 <= num2)
4251382         {
4251383           if (findstring(lp, str, len, 0) >= 0)
4251384             {
4251385               return num1;
4251386             }
4251387           //
4251388           num1++;
4251389           lp = lp->next;
4251390         }
4251391       //
4251392       fprintf (stderr, "Cannot find string \"%s\"\n", str);
4251393       //
4251394       return 0;
4251395   }
4251396   //----------------------------------------------------------------------
4251397   // Return a pointer to the specified line number.
4251398   //----------------------------------------------------------------------
4251399   line_t *
4251400   findline (num_t num)
4251401   {
4251402       line_t   *lp;
4251403       num_t     lnum;
4251404       //
4251405       if ((num < 1) || (num > lastnum))
4251406         {
4251407           fprintf (stderr, "Line number %d does not exist\n", num);
4251408           return NULL;
```

```
4251409         }
4251410       //
4251411       if (curnum <= 0)
4251412         {
4251413           curnum = 1;
4251414           curline = lines.next;
4251415         }
4251416       //
4251417       if (num == curnum)
4251418         {
4251419           return curline;
4251420         }
4251421       //
4251422       lp = curline;
4251423       lnum = curnum;
4251424       //
4251425       if (num < (curnum / 2))
4251426         {
4251427           lp = lines.next;
4251428           lnum = 1;
4251429         }
4251430       else if (num > ((curnum + lastnum) / 2))
4251431         {
4251432           lp = lines.prev;
4251433           lnum = lastnum;
4251434         }
4251435       //
4251436       while (lnum < num)
4251437         {
4251438           lp = lp->next;
4251439           lnum++;
4251440         }
4251441       //
4251442       while (lnum > num)
4251443         {
4251444           lp = lp->prev;
4251445           lnum--;
4251446         }
4251447       //
4251448       return lp;
4251449   }
4251450   //----------------------------------------------------------------------
4251451   // Set the current line number.
4251452   // Returns true if successful.
4251453   //----------------------------------------------------------------------
4251454   bool
4251455   setcurnum (num_t num)
4251456   {
```

```
4251457     line_t    *lp;
4251458     //
4251459     lp = findline (num);
4251460     if (lp == NULL)
4251461       {
4251462         return false;
4251463       }
4251464     //
4251465     curnum = num;
4251466     curline = lp;
4251467     //
4251468     return true;
4251469 }
4251470
4251471 /* END CODE */
```

102.1.12 applic/getty.c

Si veda la sezione 98.1.

```
4260001 #include <unistd.h>
4260002 #include <stdio.h>
4260003 #include <stdlib.h>
4260004 #include <signal.h>
4260005 #include <sys/wait.h>
4260006 #include <limits.h>
4260007 #include <sys/os16.h>
4260008 #include <fcntl.h>
4260009 #include <stdio.h>
4260010 //----------------------------------------------------------------------
4260011 int
4260012 main (int argc, char *argv[], char *envp[])
4260013 {
4260014     char   *device_name;
4260015     int     fdn;
4260016     char   *exec_argv[2];
4260017     char  **exec_envp;
4260018     char    buffer[BUFSIZ];
4260019     ssize_t size_read;
4260020     int     status;
4260021     //
4260022     // The first argument is mandatory and must be a console terminal.
4260023     //
4260024     device_name = argv[1];
4260025     //
4260026     // A console terminal is correctly selected (but it is not checked
4260027     // if it is a really available one).
4260028     // Set as a process group leader.
```

```
4260029     //
4260030     setpgrp ();
4260031     //
4260032     // Open the terminal, that should become the controlling terminal:
4260033     // close the standard input and open the new terminal (r/w).
4260034     //
4260035     close (0);
4260036     fdn = open (device_name, O_RDWR);
4260037     if (fdn < 0)
4260038       {
4260039         //
4260040         // Cannot open terminal. A message should appear, at least
4260041         // to the current console.
4260042         //
4260043         perror (NULL);
4260044         return (-1);
4260045       }
4260046     //
4260047     // Reset terminal device permissions and ownership.
4260048     //
4260049     status = fchown (fdn, (uid_t) 0, (gid_t) 0);
4260050     if (status != 0)
4260051       {
4260052         perror (NULL);
4260053       }
4260054     status = fchmod (fdn, 0644);
4260055     if (status != 0)
4260056       {
4260057         perror (NULL);
4260058       }
4260059     //
4260060     // The terminal is open and it should be already the controlling
4260061     // one: show '/etc/issue'. The same variable 'fdn' is used, because
4260062     // the controlling terminal will never be closed (the exit syscall
4260063     // will do it).
4260064     //
4260065     fdn = open ("/etc/issue", O_RDONLY);
4260066     if (fdn > 0)
4260067       {
4260068         //
4260069         // The file is present and is shown.
4260070         //
4260071         for (size_read = 1; size_read > 0;)
4260072           {
4260073             size_read = read (fdn, buffer, (size_t) (BUFSIZ - 1));
4260074             if (size_read < 0)
4260075               {
4260076                 break;
```

```
4260077                 }
4260078               buffer[size_read] = '\0';
4260079               printf ("%s", buffer);
4260080             }
4260081           close (fdn);
4260082         }
4260083       //
4260084       // Show the terminal.
4260085       //
4260086       printf ("This is terminal %s\n", device_name);
4260087       //
4260088       // It is time to exec login: the environment is inherited directly
4260089       // from 'init'.
4260090       //
4260091       exec_argv[0] = "login";
4260092       exec_argv[1] = NULL;
4260093       exec_envp    = envp;
4260094       execve ("/bin/login", exec_argv, exec_envp);
4260095       //
4260096       // If 'execve()' returns, it is an error.
4260097       //
4260098       exit (-1);
4260099   }
```

102.1.13 applic/init.c

Si veda la sezione 98.2.

```
4270001 #include <unistd.h>
4270002 #include <stdio.h>
4270003 #include <stdlib.h>
4270004 #include <signal.h>
4270005 #include <sys/wait.h>
4270006 #include <limits.h>
4270007 #include <sys/os16.h>
4270008 #include <fcntl.h>
4270009 #include <string.h>
4270010 //----------------------------------------------------------------------
4270011 #define RESPAWN_MAX         7
4270012 #define COMMAND_MAX       100
4270013 #define LINE_MAX         1024
4270014 //----------------------------------------------------------------------
4270015 int
4270016 main (int argc, char *argv[], char *envp[])
4270017 {
4270018     //
4270019     // 'init.c' has its own 'init.crt0.s' with a very small stack
4270020     // size. Remember to verify to have enough room for the stack.
```

```
4270021     //
4270022     pid_t  pid;
4270023     int    status;
4270024     char   *exec_argv[3];
4270025     char   *exec_envp[3];
4270026     char   buffer[LINE_MAX];
4270027     int    r;                    // Respawn table index.
4270028     int    b;                    // Buffer index.
4270029     size_t size_read;
4270030     char   *inittab_id;
4270031     char   *inittab_runlevels;
4270032     char   *inittab_action;
4270033     char   *inittab_process;
4270034     int    eof;
4270035     int    fd;
4270036     //
4270037     // It follows a table for commands to be respawn.
4270038     //
4270039     struct {
4270040         pid_t   pid;
4270041         char    command[COMMAND_MAX];
4270042     } respawn[RESPAWN_MAX];
4270043
4270044     //---------------------------------------------------------------------
4270045     signal (SIGHUP,  SIG_IGN);
4270046     signal (SIGINT,  SIG_IGN);
4270047     signal (SIGQUIT, SIG_IGN);
4270048     signal (SIGILL,  SIG_IGN);
4270049     signal (SIGABRT, SIG_IGN);
4270050     signal (SIGFPE,  SIG_IGN);
4270051 //  signal (SIGKILL, SIG_IGN);  Cannot ignore SIGKILL.
4270052     signal (SIGSEGV, SIG_IGN);
4270053     signal (SIGPIPE, SIG_IGN);
4270054     signal (SIGALRM, SIG_IGN);
4270055     signal (SIGTERM, SIG_IGN);
4270056 //  signal (SIGSTOP, SIG_IGN);  Cannot ignore SIGSTOP.
4270057     signal (SIGTSTP, SIG_IGN);
4270058     signal (SIGCONT, SIG_IGN);
4270059     signal (SIGTTIN, SIG_IGN);
4270060     signal (SIGTTOU, SIG_IGN);
4270061     signal (SIGUSR1, SIG_IGN);
4270062     signal (SIGUSR2, SIG_IGN);
4270063     //---------------------------------------------------------------------
4270064     printf ("init\n");
4270065     //  heap_clear ();
4270066     //  process_info ();
4270067     //---------------------------------------------------------------------
4270068     //
```

```
4270069      // Reset the 'respawn' table.
4270070      //
4270071      for (r = 0; r < RESPAWN_MAX; r++)
4270072        {
4270073          respawn[r].pid = 0;
4270074          respawn[r].command[0]             = 0;
4270075          respawn[r].command[COMMAND_MAX-1] = 0;
4270076        }
4270077      //
4270078      // Read the '/etc/inittab' file.
4270079      //
4270080      fd = open ("/etc/inittab", O_RDONLY);
4270081      //
4270082      if (fd < 0)
4270083        {
4270084          perror ("Cannot open file '/etc/inittab'");
4270085          exit (-1);
4270086        }
4270087      //
4270088      //
4270089      //
4270090      for (eof = 0, r = 0; !eof && r < RESPAWN_MAX; r++)
4270091        {
4270092          for (b = 0; b < LINE_MAX; b++)
4270093            {
4270094              size_read = read (fd, &buffer[b], (size_t) 1);
4270095              if (size_read <= 0)
4270096                {
4270097                  buffer[b] = 0;
4270098                  eof       = 1;           // Close the read loop.
4270099                  break;
4270100                }
4270101              if (buffer[b] == '\n')
4270102                {
4270103                  buffer[b]  = 0;
4270104                  break;
4270105                }
4270106            }
4270107          //
4270108          // Remove comments: just replace '#' with '\0'.
4270109          //
4270110          for (b = 0; b < LINE_MAX; b++)
4270111            {
4270112              if (buffer[b] == '#')
4270113                {
4270114                  buffer[b] = 0;
4270115                  break;
4270116                }
```

```
4270117             }
4270118           //
4270119           // If the buffer is an empty string, just loop to next
4270120           // record.
4270121           //
4270122           if (strlen (buffer) == 0)
4270123             {
4270124               r--;
4270125               continue;
4270126             }
4270127           //
4270128           //
4270129           //
4270130           inittab_id        = strtok (buffer, ":");
4270131           inittab_runlevels = strtok (NULL, ":");
4270132           inittab_action    = strtok (NULL, ":");
4270133           inittab_process   = strtok (NULL, ":");
4270134           //
4270135           // Only action `respawn' is used.
4270136           //
4270137           if (strcmp (inittab_action, "respawn") == 0)
4270138             {
4270139               strncpy (respawn[r].command, inittab_process, COMMAND_MAX);
4270140             }
4270141           else
4270142             {
4270143               r--;
4270144             }
4270145         }
4270146       //
4270147       //
4270148       //
4270149       close (fd);
4270150       //
4270151       // Define common environment.
4270152       //
4270153       exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
4270154       exec_envp[1] = "CONSOLE=/dev/console";
4270155       exec_envp[2] = NULL;
4270156       //
4270157       // Start processes.
4270158       //
4270159       for (r = 0; r < RESPAWN_MAX; r++)
4270160         {
4270161           if (strlen (respawn[r].command) > 0)
4270162             {
4270163               respawn[r].pid = fork ();
4270164               if (respawn[r].pid == 0)
```

```
4270165                 {
4270166                   exec_argv[0] = strtok (respawn[r].command, " \t");
4270167                   exec_argv[1] = strtok (NULL, " \t");
4270168                   exec_argv[2] = NULL;
4270169                   execve (exec_argv[0], exec_argv, exec_envp);
4270170                   perror (NULL);
4270171                   exit (0);
4270172                 }
4270173             }
4270174         }
4270175       //
4270176       // Wait for the death of child.
4270177       //
4270178       while (1)
4270179         {
4270180           pid = wait (&status);
4270181           for (r = 0; r < RESPAWN_MAX; r++)
4270182             {
4270183               if (pid == respawn[r].pid)
4270184                 {
4270185                   //
4270186                   // Run it again.
4270187                   //
4270188                   respawn[r].pid = fork ();
4270189                   if (respawn[r].pid == 0)
4270190                     {
4270191                       exec_argv[0] = strtok (respawn[r].command, " \t");
4270192                       exec_argv[1] = strtok (NULL, " \t");
4270193                       exec_argv[2] = NULL;
4270194                       execve (exec_argv[0], exec_argv, exec_envp);
4270195                       exit (0);
4270196                     }
4270197                   break;
4270198                 }
4270199             }
4270200         }
4270201   }
```

102.1.14 applic/kill.c

«

Si veda la sezione 92.10.

```
4280001   #include <sys/os16.h>
4280002   #include <sys/stat.h>
4280003   #include <sys/types.h>
4280004   #include <unistd.h>
4280005   #include <stdlib.h>
4280006   #include <fcntl.h>
```

```
4280007 #include <errno.h>
4280008 #include <signal.h>
4280009 #include <stdio.h>
4280010 #include <string.h>
4280011 #include <limits.h>
4280012 #include <libgen.h>
4280013 //----------------------------------------------------------------------
4280014 static void usage          (void);
4280015 //----------------------------------------------------------------------
4280016 int
4280017 main (int argc, char *argv[], char *envp[])
4280018 {
4280019     int           signal;
4280020     int           pid;
4280021     int           a;              // Index inside arguments.
4280022     int           option_s = 0;
4280023     int           option_l = 0;
4280024     int           opt;
4280025     extern char *optarg;
4280026     extern int   optopt;
4280027     //
4280028     // There must be at least an option, plus the program name.
4280029     //
4280030     if (argc < 2)
4280031       {
4280032         usage ();
4280033         return (1);
4280034       }
4280035     //
4280036     // Check for options.
4280037     //
4280038     while ((opt = getopt (argc, argv, ":ls:")) != -1)
4280039       {
4280040         switch (opt)
4280041           {
4280042             case 'l':
4280043                 option_l = 1;
4280044                 break;
4280045             case 's':
4280046                 option_s = 1;
4280047                 //
4280048                 // In that case, there must be at least three arguments:
4280049                 // the option, the signal and the process id.
4280050                 //
4280051                 if (argc < 4)
4280052                   {
4280053                     usage ();
4280054                     return (1);
```

```
4280055                  }
4280056                //
4280057                // Argument numbers are ok. Check the signal.
4280058                //
4280059                if (strcmp (optarg, "HUP") == 0)
4280060                  {
4280061                    signal = SIGHUP;
4280062                  }
4280063                else if (strcmp (optarg, "INT") == 0)
4280064                  {
4280065                    signal = SIGINT;
4280066                  }
4280067                else if (strcmp (optarg, "QUIT") == 0)
4280068                  {
4280069                    signal = SIGQUIT;
4280070                  }
4280071                else if (strcmp (optarg, "ILL") == 0)
4280072                  {
4280073                    signal = SIGILL;
4280074                  }
4280075                else if (strcmp (optarg, "ABRT") == 0)
4280076                  {
4280077                    signal = SIGABRT;
4280078                  }
4280079                else if (strcmp (optarg, "FPE") == 0)
4280080                  {
4280081                    signal = SIGFPE;
4280082                  }
4280083                else if (strcmp (optarg, "KILL") == 0)
4280084                  {
4280085                    signal = SIGKILL;
4280086                  }
4280087                else if (strcmp (optarg, "SEGV") == 0)
4280088                  {
4280089                    signal = SIGSEGV;
4280090                  }
4280091                else if (strcmp (optarg, "PIPE") == 0)
4280092                  {
4280093                    signal = SIGPIPE;
4280094                  }
4280095                else if (strcmp (optarg, "ALRM") == 0)
4280096                  {
4280097                    signal = SIGALRM;
4280098                  }
4280099                else if (strcmp (optarg, "TERM") == 0)
4280100                  {
4280101                    signal = SIGTERM;
4280102                  }
```

```
4280103                 else if (strcmp (optarg, "STOP") == 0)
4280104                   {
4280105                     signal = SIGSTOP;
4280106                   }
4280107                 else if (strcmp (optarg, "TSTP") == 0)
4280108                   {
4280109                     signal = SIGTSTP;
4280110                   }
4280111                 else if (strcmp (optarg, "CONT") == 0)
4280112                   {
4280113                     signal = SIGCONT;
4280114                   }
4280115                 else if (strcmp (optarg, "CHLD") == 0)
4280116                   {
4280117                     signal = SIGCHLD;
4280118                   }
4280119                 else if (strcmp (optarg, "TTIN") == 0)
4280120                   {
4280121                     signal = SIGTTIN;
4280122                   }
4280123                 else if (strcmp (optarg, "TTOU") == 0)
4280124                   {
4280125                     signal = SIGTTOU;
4280126                   }
4280127                 else if (strcmp (optarg, "USR1") == 0)
4280128                   {
4280129                     signal = SIGUSR1;
4280130                   }
4280131                 else if (strcmp (optarg, "USR2") == 0)
4280132                   {
4280133                     signal = SIGUSR2;
4280134                   }
4280135                 else
4280136                   {
4280137                     fprintf (stderr, "Unknown signal %s.\n", optarg);
4280138                     return (1);
4280139                   }
4280140                 break;
4280141             case '?':
4280142                 fprintf (stderr, "Unknown option -%c.\n", optopt);
4280143                 usage ();
4280144                 return (1);
4280145                 break;
4280146             case ':':
4280147                 fprintf (stderr, "Missing argument for option -%c\n",
4280148                          optopt);
4280149                 usage ();
4280150                 return (1);
```

```
4280151                 break;
4280152             default:
4280153                 fprintf (stderr, "Getopt problem: unknown option %c\n",
4280154                          opt);
4280155                 return (1);
4280156           }
4280157       }
4280158     //
4280159     //
4280160     //
4280161     if (option_l && option_s)
4280162       {
4280163         fprintf (stderr, "Options \"-l\" and \"-s\" together ");
4280164         fprintf (stderr, "are incompatible.\n");
4280165         usage ();
4280166         return (1);
4280167       }
4280168     //
4280169     // Option "-l".
4280170     //
4280171     if (option_l)
4280172       {
4280173         printf ("HUP ");
4280174         printf ("INT ");
4280175         printf ("QUIT ");
4280176         printf ("ILL ");
4280177         printf ("ABRT ");
4280178         printf ("FPE ");
4280179         printf ("KILL ");
4280180         printf ("SEGV ");
4280181         printf ("PIPE ");
4280182         printf ("ALRM ");
4280183         printf ("TERM ");
4280184         printf ("STOP ");
4280185         printf ("TSTP ");
4280186         printf ("CONT ");
4280187         printf ("CHLD ");
4280188         printf ("TTIN ");
4280189         printf ("TTOU ");
4280190         printf ("USR1 ");
4280191         printf ("USR2 ");
4280192         printf ("\n");
4280193       }
4280194     //
4280195     // Option "-s".
4280196     //
4280197     if (option_s)
4280198       {
```

```
4280199          //
4280200          // Scan arguments.
4280201          //
4280202          for (a = 3; a < argc; a++)
4280203            {
4280204              //
4280205              // Get PID.
4280206              //
4280207              pid = atoi (argv[a]);
4280208              if (pid > 0)
4280209                {
4280210                  //
4280211                  // Kill.
4280212                  //
4280213                  if (kill (pid, signal) < 0)
4280214                    {
4280215                      perror (argv[a]);
4280216                    }
4280217                }
4280218              else
4280219                {
4280220                  fprintf (stderr, "Invalid PID %s.", argv[a]);
4280221                }
4280222            }
4280223        }
4280224      //
4280225      // All done.
4280226      //
4280227      return (0);
4280228  }
4280229  //----------------------------------------------------------------------
4280230  static void
4280231  usage (void)
4280232  {
4280233      fprintf (stderr, "Usage: kill -s SIGNAL_NAME PID...\n");
4280234      fprintf (stderr, "       kill -l\n");
4280235  }
```

102.1.15 applic/ln.c

«

Si veda la sezione 92.11.

```
4290001  #include <sys/os16.h>
4290002  #include <sys/stat.h>
4290003  #include <sys/types.h>
4290004  #include <unistd.h>
4290005  #include <stdlib.h>
4290006  #include <fcntl.h>
```

```
4290007 #include <errno.h>
4290008 #include <signal.h>
4290009 #include <stdio.h>
4290010 #include <string.h>
4290011 #include <limits.h>
4290012 #include <libgen.h>
4290013 //----------------------------------------------------------------------
4290014 static void usage (void);
4290015 //----------------------------------------------------------------------
4290016 int
4290017 main (int argc, char *argv[], char *envp[])
4290018 {
4290019     char        *source;
4290020     char        *destination;
4290021     char        *new_destination;
4290022     struct stat file_status;
4290023     int          dest_is_a_dir = 0;
4290024     int          a;                      // Argument index.
4290025     char         path[PATH_MAX];
4290026     //
4290027     // There must be at least two arguments, plus the program name.
4290028     //
4290029     if (argc < 3)
4290030       {
4290031         usage ();
4290032         return (1);
4290033       }
4290034     //
4290035     // Select the last argument as the destination.
4290036     //
4290037     destination = argv[argc-1];
4290038     //
4290039     // Check if it is a directory and save it in a flag.
4290040     //
4290041     if (stat (destination, &file_status) == 0)
4290042       {
4290043         if (S_ISDIR (file_status.st_mode))
4290044           {
4290045             dest_is_a_dir = 1;
4290046           }
4290047       }
4290048     //
4290049     // If there are more than two arguments, verify that the last
4290050     // one is a directory.
4290051     //
4290052     if (argc > 3)
4290053       {
4290054         if (!dest_is_a_dir)
```

```
4290055             {
4290056               usage ();
4290057               fprintf (stderr, "The destination \"%s\" ",
4290058                        destination);
4290059               fprintf (stderr, "is not a directory!\n");
4290060               return (1);
4290061             }
4290062         }
4290063       //
4290064       // Scan the arguments, excluded the last, that is the destination.
4290065       //
4290066       for (a = 1; a < (argc - 1); a++)
4290067         {
4290068           //
4290069           // Source.
4290070           //
4290071           source = argv[a];
4290072           //
4290073           // Verify access permissions.
4290074           //
4290075           if (access (source, R_OK) < 0)
4290076             {
4290077               perror (source);
4290078               continue;
4290079             }
4290080           //
4290081           // Destination.
4290082           //
4290083           // If it is a directory, the destination path
4290084           // must be corrected.
4290085           //
4290086           if (dest_is_a_dir)
4290087             {
4290088               path[0] = 0;
4290089               strcat (path, destination);
4290090               strcat (path, "/");
4290091               strcat (path, basename (source));
4290092               //
4290093               // Update the destination path.
4290094               //
4290095               new_destination = path;
4290096             }
4290097           else
4290098             {
4290099               new_destination = destination;
4290100             }
4290101           //
4290102           // Check if destination file exists.
```

```
4290103             //
4290104             if (stat (new_destination, &file_status) == 0)
4290105               {
4290106                 fprintf (stderr, "The destination file, \"%s\", ",
4290107                          new_destination);
4290108                 fprintf (stderr, "already exists!\n");
4290109                 continue;
4290110               }
4290111             //
4290112             // Everything is ready for the link.
4290113             //
4290114             if (link (source, new_destination) < 0)
4290115               {
4290116                 perror (new_destination);
4290117                 continue;
4290118               }
4290119           }
4290120         //
4290121         // All done.
4290122         //
4290123         return (0);
4290124 }
4290125 //----------------------------------------------------------------------
4290126 static void
4290127 usage (void)
4290128 {
4290129     fprintf (stderr, "Usage: ln OLD_NAME NEW_NAME\n");
4290130     fprintf (stderr, "       ln FILE... DIRECTORY\n");
4290131 }
```

102.1.16 applic/login.c

Si veda la sezione 92.12.

```
4300001 #include <unistd.h>
4300002 #include <stdlib.h>
4300003 #include <sys/stat.h>
4300004 #include <sys/types.h>
4300005 #include <fcntl.h>
4300006 #include <errno.h>
4300007 #include <unistd.h>
4300008 #include <signal.h>
4300009 #include <stdio.h>
4300010 #include <sys/wait.h>
4300011 #include <stdio.h>
4300012 #include <string.h>
4300013 #include <limits.h>
4300014 #include <stdint.h>
```

```
4300015 #include <sys/os16.h>
4300016 //----------------------------------------------------------------------
4300017 #define LOGIN_MAX       64
4300018 #define PASSWORD_MAX    64
4300019 #define HOME_MAX        64
4300020 #define LINE_MAX        1024
4300021 //----------------------------------------------------------------------
4300022 int
4300023 main (int argc, char *argv[], char *envp[])
4300024 {
4300025     char    login[LOGIN_MAX];
4300026     char    password[PASSWORD_MAX];
4300027     char    buffer[LINE_MAX];
4300028     char   *user_name;
4300029     char   *user_password;
4300030     char   *user_uid;
4300031     char   *user_gid;
4300032     char   *user_description;
4300033     char   *user_home;
4300034     char   *user_shell;
4300035     uid_t   uid;
4300036     uid_t   euid;
4300037     int     fd;
4300038     ssize_t size_read;
4300039     int     b;          // Index inside buffer.
4300040     int     loop;
4300041     char   *exec_argv[2];
4300042     int     status;
4300043     char   *tty_path;
4300044     //
4300045     // Check if login is running correctly.
4300046     //
4300047     euid = geteuid ();
4300048     uid  = geteuid ();
4300049     //  //
4300050     //  // Show process info.
4300051     //  //
4300052     //  heap_clear ();
4300053     //  process_info ();
4300054     //
4300055     // Check privileges.
4300056     //
4300057     if (!(uid == 0 && euid == 0))
4300058       {
4300059         printf ("%s: can only run with root privileges!\n", argv[0]);
4300060         exit (-1);
4300061       }
4300062     //
```

```
4300063     // Prepare arguments for the shell call.
4300064     //
4300065     exec_argv[0] = "-";
4300066     exec_argv[1] = NULL;
4300067     //
4300068     // Login.
4300069     //
4300070     while (1)
4300071       {
4300072         fd = open ("/etc/passwd", O_RDONLY);
4300073         //
4300074         if (fd < 0)
4300075           {
4300076             perror ("Cannot open file `/etc/passwd'");
4300077             exit (-1);
4300078           }
4300079         //
4300080         printf ("Log in as \"root\" or \"user\" "
4300081                 "with password \"ciao\" :-)\n");
4300082         input_line (login, "login:", LOGIN_MAX, INPUT_LINE_ECHO);
4300083         //
4300084         //
4300085         //
4300086         loop = 1;
4300087         while (loop)
4300088           {
4300089             for (b = 0; b < LINE_MAX; b++)
4300090               {
4300091                 size_read = read (fd, &buffer[b], (size_t) 1);
4300092                 if (size_read <= 0)
4300093                   {
4300094                     buffer[b] = 0;
4300095                     loop = 0;           // Close the middle loop.
4300096                     break;
4300097                   }
4300098                 if (buffer[b] == '\n')
4300099                   {
4300100                     buffer[b] = 0;
4300101                     break;
4300102                   }
4300103               }
4300104             //
4300105             user_name        = strtok (buffer, ":");
4300106             user_password    = strtok (NULL, ":");
4300107             user_uid         = strtok (NULL, ":");
4300108             user_gid         = strtok (NULL, ":");
4300109             user_description = strtok (NULL, ":");
4300110             user_home        = strtok (NULL, ":");
```

```
4300111                 user_shell       = strtok (NULL, ":");
4300112                 //
4300113                 if (strcmp (user_name, login) == 0)
4300114                   {
4300115                     input_line (password, "password:", PASSWORD_MAX,
4300116                                 INPUT_LINE_STARS);
4300117                     //
4300118                     // Compare passwords: empty passwords are not allowed.
4300119                     //
4300120                     if (strcmp (user_password, password) == 0)
4300121                       {
4300122                         uid  = atoi (user_uid);
4300123                         euid = uid;
4300124                         //
4300125                         // Find the controlling terminal and change
4300126                         // property and access permissions.
4300127                         //
4300128                         tty_path = ttyname (STDIN_FILENO);
4300129                         if (tty_path != NULL)
4300130                           {
4300131                             status = chown (tty_path, uid, 0);
4300132                             if (status != 0)
4300133                               {
4300134                                 perror (NULL);
4300135                               }
4300136                             status = chmod (tty_path, 0600);
4300137                             if (status != 0)
4300138                               {
4300139                                 perror (NULL);
4300140                               }
4300141                           }
4300142                         //
4300143                         // Cd to the home directory, if present.
4300144                         //
4300145                         status = chdir (user_home);
4300146                         if (status != 0)
4300147                           {
4300148                             perror (NULL);
4300149                           }
4300150                         //
4300151                         // Now change personality.
4300152                         //
4300153                         setuid (uid);
4300154                         seteuid (euid);
4300155                         //
4300156                         // Run the shell, replacing the login process; the
4300157                         // environment is taken from 'init'.
4300158                         //
```

```
4300159                         execve (user_shell, exec_argv, envp);
4300160                         exit (0);
4300161                       }
4300162                     //
4300163                     // Login failed: will try again.
4300164                     //
4300165                     loop = 0;               // Close the middle loop.
4300166                     break;
4300167                   }
4300168               }
4300169             close (fd);
4300170         }
4300171 }
```

102.1.17 applic/ls.c

Si veda la sezione 92.13.

```
4310001 #include <sys/os16.h>
4310002 #include <sys/stat.h>
4310003 #include <sys/types.h>
4310004 #include <unistd.h>
4310005 #include <stdlib.h>
4310006 #include <fcntl.h>
4310007 #include <errno.h>
4310008 #include <signal.h>
4310009 #include <stdio.h>
4310010 #include <string.h>
4310011 #include <limits.h>
4310012 #include <libgen.h>
4310013 #include <dirent.h>
4310014 #include <pwd.h>
4310015 #include <time.h>
4310016 
4310017 #define BUFFER_SIZE     16384
4310018 #define LIST_SIZE       256
4310019 
4310020 //----------------------------------------------------------------------
4310021 static void usage         (void);
4310022 //----------------------------------------------------------------------
4310023 //----------------------------------------------------------------------
4310024 int compare (void *p1, void *p2);
4310025 //----------------------------------------------------------------------
4310026 int
4310027 main (int argc, char *argv[], char *envp[])
4310028 {
4310029     int             option_a = 0;
4310030     int             option_l = 0;
```

```
4310031     int              opt;
4310032 //  extern char     *optarg;              // not used.
4310033     extern int       optind;
4310034     extern int       optopt;
4310035     struct stat      file_status;
4310036     DIR             *dp;
4310037     struct dirent   *dir;
4310038     char             buffer[BUFFER_SIZE];
4310039     int              b;                    // Buffer index.
4310040     char            *list[LIST_SIZE];
4310041     int              l;                    // List index.
4310042     int              len;                  // Name length.
4310043     char            *path = NULL;
4310044     char             pathname[PATH_MAX];
4310045     struct passwd   *pws;
4310046     struct tm       *tms;
4310047     //
4310048     // Check for options.
4310049     //
4310050     while ((opt = getopt (argc, argv, ":al")) != -1)
4310051       {
4310052         switch (opt)
4310053           {
4310054             case 'l':
4310055                 option_l = 1;
4310056                 break;
4310057             case 'a':
4310058                 option_a = 1;
4310059                 break;
4310060             case '?':
4310061                 fprintf (stderr, "Unknown option -%c.\n", optopt);
4310062                 usage ();
4310063                 return (1);
4310064                 break;
4310065             case ':':
4310066                 fprintf (stderr, "Missing argument for option -%c\n",
4310067                          optopt);
4310068                 usage ();
4310069                 return (1);
4310070                 break;
4310071             default:
4310072                 fprintf (stderr, "Getopt problem: unknown option %c\n",
4310073                          opt);
4310074                 return (1);
4310075           }
4310076       }
4310077     //
4310078     // If no arguments are present, at least the current directory is
```

```
4310079     // read.
4310080     //
4310081     if (optind == argc)
4310082       {
4310083         //
4310084         // There are no more arguments. Replace the program name,
4310085         // corresponding to 'argv[0]', with the current directory
4310086         // path string.
4310087         //
4310088         argv[0] = ".";
4310089         argc    = 1;
4310090         optind  = 0;
4310091       }
4310092     //
4310093     // This is a very simplified 'ls': if there is only a name
4310094     // and it is a directory, the directory content is taken as
4310095     // the new 'argv[]' array.
4310096     //
4310097     if (optind == (argc -1))
4310098       {
4310099         //
4310100         // There is a request for a single name. Test if it exists
4310101         // and if it is a directory.
4310102         //
4310103         if (stat(argv[optind], &file_status) != 0)
4310104           {
4310105             fprintf (stderr, "File \"%s\" does not exist!\n",
4310106                      argv[optind]);
4310107             return (2);
4310108           }
4310109         //
4310110         if (S_ISDIR (file_status.st_mode))
4310111           {
4310112             //
4310113             // Save the directory inside the 'path' pointer.
4310114             //
4310115             path = argv[optind];
4310116             //
4310117             // Open the directory.
4310118             //
4310119             dp = opendir (argv[optind]);
4310120             if (dp == NULL)
4310121               {
4310122                 perror (argv[optind]);
4310123                 return (3);
4310124               }
4310125             //
4310126             // Read the directory and fill the buffer with names.
```

```
4310127             //
4310128             b = 0;
4310129             l = 0;
4310130             while ((dir = readdir (dp)) != NULL)
4310131               {
4310132                 len = strlen (dir->d_name);
4310133                 //
4310134                 // Check if the buffer can hold it.
4310135                 //
4310136                 if ((b + len + 1 ) > BUFFER_SIZE)
4310137                   {
4310138                     fprintf (stderr, "not enough memory\n");
4310139                     break;
4310140                   }
4310141                 //
4310142                 // Consider the directory item only if there is
4310143                 // a valid name. If it is empty, just ignore it.
4310144                 //
4310145                 if (len > 0)
4310146                   {
4310147                     strcpy (&buffer[b], dir->d_name);
4310148                     list[l] = &buffer[b];
4310149                     b += len + 1;
4310150                     l++;
4310151                   }
4310152               }
4310153             //
4310154             // Close the directory.
4310155             //
4310156             closedir (dp);
4310157             //
4310158             // Sort the list.
4310159             //
4310160             qsort (list, (size_t) l, sizeof (char *), compare);
4310161
4310162             //
4310163             // Convert the directory list into a new 'argv[]' array,
4310164             // with a valid 'argc'. The variable 'optind' must be
4310165             // reset to the first element index, because there is
4310166             // no program name inside the new 'argv[]' at index zero.
4310167             //
4310168             argv   = list;
4310169             argc   = l;
4310170             optind = 0;
4310171           }
4310172       }
4310173     //
4310174     // Scan arguments, or list converted into 'argv[]'.
```

```
4310175      //
4310176      for (; optind < argc; optind++)
4310177        {
4310178          if (argv[optind][0] == '.')
4310179            {
4310180              //
4310181              // Current name starts with `.'.
4310182              //
4310183              if (!option_a)
4310184                {
4310185                  //
4310186                  // Do not show name starting with `.'.
4310187                  //
4310188                  continue;
4310189                }
4310190            }
4310191          //
4310192          // Build the pathname.
4310193          //
4310194          if (path == NULL)
4310195            {
4310196              strcpy (&pathname[0], argv[optind]);
4310197            }
4310198          else
4310199            {
4310200              strcpy (pathname, path);
4310201              strcat (pathname, "/");
4310202              strcat (pathname, argv[optind]);
4310203            }
4310204          //
4310205          // Check if file exists, reading status.
4310206          //
4310207          if (stat(pathname, &file_status) != 0)
4310208            {
4310209              fprintf (stderr, "File \"%s\" does not exist!\n",
4310210                       pathname);
4310211              return (2);
4310212            }
4310213          //
4310214          // Show file name.
4310215          //
4310216          if (option_l)
4310217            {
4310218              //
4310219              // Print the file type.
4310220              //
4310221              if      (S_ISBLK  (file_status.st_mode)) printf ("b");
4310222              else if (S_ISCHR  (file_status.st_mode)) printf ("c");
```

```
4310223                 else if (S_ISFIFO (file_status.st_mode)) printf ("p");
4310224                 else if (S_ISREG  (file_status.st_mode)) printf ("-");
4310225                 else if (S_ISDIR  (file_status.st_mode)) printf ("d");
4310226                 else if (S_ISLNK  (file_status.st_mode)) printf ("l");
4310227                 else if (S_ISSOCK (file_status.st_mode)) printf ("s");
4310228                 else                                     printf ("?");
4310229                 //
4310230                 // Print permissions.
4310231                 //
4310232                 if (S_IRUSR & file_status.st_mode) printf ("r");
4310233                 else printf ("-");
4310234                 if (S_IWUSR & file_status.st_mode) printf ("w");
4310235                 else printf ("-");
4310236                 if (S_IXUSR & file_status.st_mode) printf ("x");
4310237                 else printf ("-");
4310238                 if (S_IRGRP & file_status.st_mode) printf ("r");
4310239                 else printf ("-");
4310240                 if (S_IWGRP & file_status.st_mode) printf ("w");
4310241                 else printf ("-");
4310242                 if (S_IXGRP & file_status.st_mode) printf ("x");
4310243                 else printf ("-");
4310244                 if (S_IROTH & file_status.st_mode) printf ("r");
4310245                 else printf ("-");
4310246                 if (S_IWOTH & file_status.st_mode) printf ("w");
4310247                 else printf ("-");
4310248                 if (S_IXOTH & file_status.st_mode) printf ("x");
4310249                 else printf ("-");
4310250                 //
4310251                 // Print links.
4310252                 //
4310253                 printf (" %3i", (int) file_status.st_nlink);
4310254                 //
4310255                 // Print owner.
4310256                 //
4310257                 pws = getpwuid (file_status.st_uid);
4310258                 //
4310259                 printf (" %s", pws->pw_name);
4310260                 //
4310261                 // Print group (no group available);
4310262                 //
4310263                 printf (" (no group)");
4310264                 //
4310265                 // Print file size or device major-minor.
4310266                 //
4310267                 if (S_ISBLK (file_status.st_mode)
4310268                     || S_ISCHR (file_status.st_mode))
4310269                   {
4310270                     printf (" %3i,", (int) major (file_status.st_rdev));
```

```
4310271                 printf (" %3i",  (int) minor (file_status.st_rdev));
4310272               }
4310273             else
4310274               {
4310275                 printf (" %8i", (int) file_status.st_size);
4310276               }
4310277             //
4310278             // Print modification date and time.
4310279             //
4310280             tms = localtime (&(file_status.st_mtime));
4310281             printf (" %4u-%02u-%02u %02u:%02u",
4310282                     tms->tm_year, tms->tm_mon, tms->tm_mday,
4310283                     tms->tm_hour, tms->tm_min);
4310284             //
4310285             // Print file name, but with no additional path.
4310286             //
4310287             printf (" %s\n", argv[optind]);
4310288           }
4310289         else
4310290           {
4310291             //
4310292             // Just show the file name and go to the next line.
4310293             //
4310294             printf ("%s\n", argv[optind]);
4310295           }
4310296       }
4310297     //
4310298     // All done.
4310299     //
4310300     return (0);
4310301 }
4310302 //----------------------------------------------------------------------
4310303 static void
4310304 usage (void)
4310305 {
4310306     fprintf (stderr, "Usage: ls [OPTION] [FILE]...\n");
4310307 }
4310308 //----------------------------------------------------------------------
4310309 int
4310310 compare (void *p1, void *p2)
4310311 {
4310312     char **pp1 = p1;
4310313     char **pp2 = p2;
4310314     //
4310315     return (strcmp (*pp1, *pp2));
4310316 }
4310317
```

102.1.18 applic/man.c

«

Si veda la sezione 92.14.

```
4320001 #include <unistd.h>
4320002 #include <stdlib.h>
4320003 #include <errno.h>
4320004 //----------------------------------------------------------------------
4320005 #define MAX_LINES   20
4320006 #define MAX_COLUMNS 80
4320007 //----------------------------------------------------------------------
4320008 static char *man_page_directory = "/usr/share/man";
4320009 //----------------------------------------------------------------------
4320010 static void  usage          (void);
4320011 static FILE *open_man_page  (int section, char *name);
4320012 static void build_path_name (int section, char *name, char *path);
4320013 //----------------------------------------------------------------------
4320014 int
4320015 main (int argc, char *argv[], char *envp[])
4320016 {
4320017     FILE       *fp;
4320018     char       *name;
4320019     int         section;
4320020     int         c;
4320021     int         line   = 1; // Line internal counter.
4320022     int         column = 1; // Column internal counter.
4320023     int         loop;
4320024     //
4320025     // There must be minimum an argument, and maximum two.
4320026     //
4320027     if (argc < 2 || argc > 3)
4320028       {
4320029         usage ();
4320030         return (1);
4320031       }
4320032     //
4320033     // If there are two arguments, there must be the
4320034     // section number.
4320035     //
4320036     if (argc == 3)
4320037       {
4320038         section = atoi (argv[1]);
4320039         name = argv[2];
4320040       }
4320041     else
4320042       {
4320043         section = 0;
4320044         name = argv[1];
4320045       }
```

```
4320046       //
4320047       // Try to open the manual page.
4320048       //
4320049       fp = open_man_page (section, name);
4320050       //
4320051       if (fp == NULL)
4320052         {
4320053           //
4320054           // Error opening file.
4320055           //
4320056           return (1);
4320057         }
4320058
4320059       //
4320060       // The following loop continues while the file
4320061       // gives characters, or when a command to change
4320062       // file or to quit is given.
4320063       //
4320064       for (loop = 1; loop; )
4320065         {
4320066           //
4320067           // Read a single character.
4320068           //
4320069           c = getc (fp);
4320070           //
4320071           if (c == EOF)
4320072             {
4320073               loop = 0;
4320074               break;
4320075             }
4320076           //
4320077           // If the character read is a special one,
4320078           // the line/column calculation is modified,
4320079           // so that it is known when to stop scrolling.
4320080           //
4320081           switch (c)
4320082             {
4320083               case '\r':
4320084                   //
4320085                   // Displaying this character, the cursor should go
4320086                   // back to the first column. So the column counter
4320087                   // is reset.
4320088                   //
4320089                   column = 1;
4320090                   break;
4320091               case '\n':
4320092                   //
4320093                   // Displaying this character, the cursor should go
```

```
4320094                     // back to the next line, at the first column.
4320095                     // So the column counter is reset and the line
4320096                     // counter is incremented.
4320097                     //
4320098                     line++;
4320099                     column = 1;
4320100                     break;
4320101                 case '\b':
4320102                     //
4320103                     // Displaying this character, the cursor should go
4320104                     // back one position, unless it is already at the
4320105                     // beginning.
4320106                     //
4320107                     if (column > 1)
4320108                       {
4320109                         column--;
4320110                       }
4320111                     break;
4320112                 default:
4320113                     //
4320114                     // Any other character must increase the column
4320115                     // counter.
4320116                     //
4320117                     column++;
4320118               }
4320119             //
4320120             // Display the character, even if it is a special one:
4320121             // it is responsibility of the screen device management
4320122             // to do something good with special characters.
4320123             //
4320124             putchar (c);
4320125             //
4320126             // If the column counter is gone beyond the screen columns,
4320127             // then adjust the column counter and increment the line
4320128             // counter.
4320129             //
4320130             if (column > MAX_COLUMNS)
4320131               {
4320132                 column -= MAX_COLUMNS;
4320133                 line++;
4320134               }
4320135             //
4320136             // Check if there is space for scrolling.
4320137             //
4320138             if (line < MAX_LINES)
4320139               {
4320140                 continue;
4320141               }
```

```
4320142          //
4320143          // Here, displayed lines are MAX_LINES.
4320144          //
4320145          if (column > 1)
4320146            {
4320147              //
4320148              // Something was printed at the current line: must
4320149              // do a new line.
4320150              //
4320151              putchar ('\n');
4320152            }
4320153          //
4320154          // Show the more prompt.
4320155          //
4320156          printf ("--More--");
4320157          fflush (stdout);
4320158          //
4320159          // Read a character from standard input.
4320160          //
4320161          c = getchar ();
4320162          //
4320163          // Consider command `q', but any other character
4320164          // can be introduced, to let show the next page.
4320165          //
4320166          switch (c)
4320167            {
4320168              case 'Q':
4320169              case 'q':
4320170                  //
4320171                  // Quit. But must erase the `--More--' prompt.
4320172                  //
4320173                  printf ("\b \b\b \b\b \b\b \b\b \b");
4320174                  printf ("\b \b\b \b\b \b\b \b");
4320175                  fclose (fp);
4320176                  return (0);
4320177            }
4320178          //
4320179          // Backspace to overwrite `--More--' and the character
4320180          // pressed.
4320181          //
4320182          printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
4320183          //
4320184          // Reset line/column counters.
4320185          //
4320186          column = 1;
4320187          line = 1;
4320188        }
4320189      //
```

```
4320190     // Close the file pointer if it is still open.
4320191     //
4320192     if (fp != NULL)
4320193       {
4320194         fclose (fp);
4320195       }
4320196     //
4320197     return (0);
4320198 }
4320199 //----------------------------------------------------------------------
4320200 static void
4320201 usage (void)
4320202 {
4320203     fprintf (stderr, "Usage: man [SECTION] NAME\n");
4320204 }
4320205 //----------------------------------------------------------------------
4320206 FILE *
4320207 open_man_page (int section, char *name)
4320208 {
4320209     FILE        *fp;
4320210     char         path[PATH_MAX];
4320211     struct stat file_status;
4320212     //
4320213     //
4320214     //
4320215     if (section > 0)
4320216       {
4320217         build_path_name (section, name, path);
4320218         //
4320219         // Check if file exists.
4320220         //
4320221         if (stat (path, &file_status) != 0)
4320222           {
4320223             fprintf (stderr, "Man page %s(%i) does not exist!\n",
4320224                      name, section);
4320225             return (NULL);
4320226           }
4320227       }
4320228     else
4320229       {
4320230         //
4320231         // Must try a section.
4320232         //
4320233         for (section = 1; section < 9; section++)
4320234           {
4320235             build_path_name (section, name, path);
4320236             //
4320237             // Check if file exists.
```

```
4320238                //
4320239                if (stat (path, &file_status) == 0)
4320240                  {
4320241                    //
4320242                    // Found.
4320243                    //
4320244                    break;
4320245                  }
4320246              }
4320247          }
4320248        //
4320249        // Check if a file was found.
4320250        //
4320251        if (section < 9)
4320252          {
4320253            fp = fopen (path, "r");
4320254            //
4320255            if (fp == NULL)
4320256              {
4320257                //
4320258                // Error opening file.
4320259                //
4320260                perror (path);
4320261                return (NULL);
4320262              }
4320263            else
4320264              {
4320265                //
4320266                // Opened right.
4320267                //
4320268                return (fp);
4320269              }
4320270          }
4320271        else
4320272          {
4320273            fprintf (stderr, "Man page %s does not exist!\n",
4320274                     name);
4320275            return (NULL);
4320276          }
4320277  }
4320278  //----------------------------------------------------------------------
4320279  void
4320280  build_path_name (int section, char *name, char *path)
4320281  {
4320282      char  string_section[10];
4320283      //
4320284      // Convert the section number into a string.
4320285      //
```

```
4320286     sprintf (string_section, "%i", section);
4320287     //
4320288     // Prepare the path to the man file.
4320289     //
4320290     path[0] = 0;
4320291     strcat (path, man_page_directory);
4320292     strcat (path, "/");
4320293     strcat (path, name);
4320294     strcat (path, ".");
4320295     strcat (path, string_section);
4320296 }
```

102.1.19 applic/mkdir.c

Si veda la sezione 92.15.

```
4330001 #include <sys/os16.h>
4330002 #include <sys/stat.h>
4330003 #include <sys/types.h>
4330004 #include <unistd.h>
4330005 #include <stdlib.h>
4330006 #include <fcntl.h>
4330007 #include <errno.h>
4330008 #include <signal.h>
4330009 #include <stdio.h>
4330010 #include <string.h>
4330011 #include <limits.h>
4330012 #include <libgen.h>
4330013 //----------------------------------------------------------------------
4330014 static int  mkdir_parents (const char *path, mode_t mode);
4330015 static void usage          (void);
4330016 //----------------------------------------------------------------------
4330017 int
4330018 main (int argc, char *argv[], char *envp[])
4330019 {
4330020     sysmsg_uarea_t msg;
4330021     int            status;
4330022     mode_t         mode     = 0;
4330023     int            m;             // Index inside mode argument.
4330024     int            digit;
4330025     char          **dir;
4330026     int            d;             // Directory index.
4330027     int            option_p = 0;
4330028     int            option_m = 0;
4330029     int            opt;
4330030     extern char   *optarg;
4330031     extern int     optind;
4330032     extern int     optopt;
```

```
4330033     //
4330034     // There must be at least an argument, plus the program name.
4330035     //
4330036     if (argc < 2)
4330037       {
4330038         usage ();
4330039         return (1);
4330040       }
4330041     //
4330042     // Check for options, starting from 'p'. The 'dir' pointer is used
4330043     // to calculate the argument pointer to the first directory [1].
4330044     // The macro-instruction 'max()' is declared inside <sys/os16.h>
4330045     // and does the expected thing.
4330046     //
4330047     while ((opt = getopt (argc, argv, ":pm:")) != -1)
4330048       {
4330049         switch (opt)
4330050           {
4330051             case 'm':
4330052                 option_m = 1;
4330053                 for (m = 0; m < strlen (optarg); m++)
4330054                   {
4330055                     digit = (optarg[m] - '0');
4330056                     if (digit < 0 || digit > 7)
4330057                       {
4330058                         usage ();
4330059                         return (2);
4330060                       }
4330061                     mode = mode * 8 + digit;
4330062                   }
4330063                 break;
4330064             case 'p':
4330065                 option_p = 1;
4330066                 break;
4330067             case '?':
4330068                 printf ("Unknown option -%c.\n", optopt);
4330069                 usage ();
4330070                 return (1);
4330071                 break;
4330072             case ':':
4330073                 printf ("Missing argument for option -%c\n", optopt);
4330074                 usage ();
4330075                 return (2);
4330076                 break;
4330077             default:
4330078                 printf ("Getopt problem: unknown option %c\n", opt);
4330079                 return (3);
4330080           }
```

```
4330081         }
4330082     //
4330083     dir = argv + optind;
4330084     //
4330085     // Check if the mode is to be set to a default value.
4330086     //
4330087     if (!option_m)
4330088       {
4330089         //
4330090         // Default mode.
4330091         //
4330092         sys (SYS_UAREA, &msg, (sizeof msg));
4330093         mode = 0777 & ~msg.umask;
4330094       }
4330095     //
4330096     // Directory creation.
4330097     //
4330098     for (d = 0; dir[d] != NULL; d++)
4330099       {
4330100         if (option_p)
4330101           {
4330102             status = mkdir_parents (dir[d], mode);
4330103             if (status != 0)
4330104              {
4330105                 perror (dir[d]);
4330106                 return (3);
4330107               }
4330108           }
4330109         else
4330110           {
4330111             status = mkdir (dir[d], mode);
4330112             if (status != 0)
4330113               {
4330114                 perror (dir[d]);
4330115                 return (4);
4330116               }
4330117           }
4330118       }
4330119     //
4330120     // All done.
4330121     //
4330122     return (0);
4330123 }
4330124 //----------------------------------------------------------------------
4330125 static int
4330126 mkdir_parents (const char *path, mode_t mode)
4330127 {
4330128     char        path_copy[PATH_MAX];
```

```
4330129     char       *path_parent;
4330130     struct stat fst;
4330131     int         status;
4330132     //
4330133     // Check if the path is empty.
4330134     //
4330135     if (path == NULL || strlen (path) == 0)
4330136       {
4330137         //
4330138         // Recursion ends here.
4330139         //
4330140         return (0);
4330141       }
4330142     //
4330143     // Check if it does already exists.
4330144     //
4330145     status = stat (path, &fst);
4330146     if (status == 0 && fst.st_mode & S_IFDIR)
4330147       {
4330148         //
4330149         // The path exists and is a directory.
4330150         //
4330151         return (0);
4330152       }
4330153     else if (status == 0 && !(fst.st_mode & S_IFDIR))
4330154       {
4330155         //
4330156         // The path exists but is not a directory.
4330157         //
4330158         errno = ENOTDIR;         // Not a directory.
4330159         return (-1);
4330160       }
4330161     //
4330162     // Get the directory path.
4330163     //
4330164     strncpy (path_copy, path, PATH_MAX);
4330165     path_parent = dirname (path_copy);
4330166     //
4330167     // If it is '.', or '/', the recursion is terminated.
4330168     //
4330169     if (strncmp (path_parent, ".", PATH_MAX) == 0  ||
4330170         strncmp (path_parent, "/", PATH_MAX) == 0)
4330171       {
4330172         return (0);
4330173       }
4330174     //
4330175     // Otherwise, continue the recursion.
4330176     //
```

```
4330177     status = mkdir_parents (path_parent, mode);
4330178     if (status != 0)
4330179       {
4330180         return (-1);
4330181       }
4330182     //
4330183     // Previous directories are there: create the current one.
4330184     //
4330185     status = mkdir (path, mode);
4330186     if (status)
4330187       {
4330188         perror (path);
4330189         return (-1);
4330190       }
4330191 
4330192     return (0);
4330193 }
4330194 //----------------------------------------------------------------------
4330195 static void
4330196 usage (void)
4330197 {
4330198     fprintf (stderr, "Usage: mkdir [-p] [-m OCTAL_MODE] DIR...\n");
4330199 }
```

102.1.20 applic/more.c

Si veda la sezione 92.16.

```
4340001 #include <unistd.h>
4340002 #include <errno.h>
4340003 //----------------------------------------------------------------------
4340004 #define MAX_LINES   20
4340005 #define MAX_COLUMNS 80
4340006 //----------------------------------------------------------------------
4340007 static void usage (void);
4340008 //----------------------------------------------------------------------
4340009 int
4340010 main (int argc, char *argv[], char *envp[])
4340011 {
4340012     FILE   *fp;
4340013     char   *name;
4340014     int     c;
4340015     int     line   = 1; // Line internal counter.
4340016     int     column = 1; // Column internal counter.
4340017     int     a;          // Index inside arguments.
4340018     int     loop;
4340019     //
4340020     // There must be at least an argument, plus the program name.
```

```
4340021     //
4340022     if (argc < 2)
4340023       {
4340024         usage ();
4340025         return (1);
4340026       }
4340027     //
4340028     // No options are allowed.
4340029     //
4340030     for (a = 1; a < argc ; a++)
4340031       {
4340032         //
4340033         // Get next name from arguments.
4340034         //
4340035         name = argv[a];
4340036         //
4340037         // Try to open the file, read only.
4340038         //
4340039         fp = fopen (name, "r");
4340040         //
4340041         if (fp == NULL)
4340042           {
4340043             //
4340044             // Error opening file.
4340045             //
4340046             perror (name);
4340047             return (1);
4340048           }
4340049         //
4340050         // Print the file name to be displayed.
4340051         //
4340052         printf ("== %s ==\n", name);
4340053         line++;
4340054         //
4340055         // The following loop continues while the file
4340056         // gives characters, or when a command to change
4340057         // file or to quit is given.
4340058         //
4340059         for (loop = 1; loop; )
4340060           {
4340061             //
4340062             // Read a single character.
4340063             //
4340064             c = getc (fp);
4340065             //
4340066             if (c == EOF)
4340067               {
4340068                 loop = 0;
```

```
4340069                  break;
4340070                }
4340071              //
4340072              // If the character read is a special one,
4340073              // the line/column calculation is modified,
4340074              // so that it is known when to stop scrolling.
4340075              //
4340076              switch (c)
4340077                {
4340078                  case '\r':
4340079                      //
4340080                      // Displaying this character, the cursor should go
4340081                      // back to the first column. So the column counter
4340082                      // is reset.
4340083                      //
4340084                      column = 1;
4340085                      break;
4340086                  case '\n':
4340087                      //
4340088                      // Displaying this character, the cursor should go
4340089                      // back to the next line, at the first column.
4340090                      // So the column counter is reset and the line
4340091                      // counter is incremented.
4340092                      //
4340093                      line++;
4340094                      column = 1;
4340095                      break;
4340096                  case '\b':
4340097                      //
4340098                      // Displaying this character, the cursor should go
4340099                      // back one position, unless it is already at the
4340100                      // beginning.
4340101                      //
4340102                      if (column > 1)
4340103                        {
4340104                          column--;
4340105                        }
4340106                      break;
4340107                  default:
4340108                      //
4340109                      // Any other character must increase the column
4340110                      // counter.
4340111                      //
4340112                      column++;
4340113                }
4340114              //
4340115              // Display the character, even if it is a special one:
4340116              // it is responsibility of the screen device management
```

```
4340117                 // to do something good with special characters.
4340118                 //
4340119                 putchar (c);
4340120                 //
4340121                 // If the column counter is gone beyond the screen columns,
4340122                 // then adjust the column counter and increment the line
4340123                 // counter.
4340124                 //
4340125                 if (column > MAX_COLUMNS)
4340126                   {
4340127                     column -= MAX_COLUMNS;
4340128                     line++;
4340129                   }
4340130                 //
4340131                 // Check if there is space for scrolling.
4340132                 //
4340133                 if (line < MAX_LINES)
4340134                   {
4340135                     continue;
4340136                   }
4340137                 //
4340138                 // Here, displayed lines are MAX_LINES.
4340139                 //
4340140                 if (column > 1)
4340141                   {
4340142                     //
4340143                     // Something was printed at the current line: must
4340144                     // do a new line.
4340145                     //
4340146                     putchar ('\n');
4340147                   }
4340148                 //
4340149                 // Show the more prompt.
4340150                 //
4340151                 printf ("--More--");
4340152                 fflush (stdout);
4340153                 //
4340154                 // Read a character from standard input.
4340155                 //
4340156                 c = getchar ();
4340157                 //
4340158                 // Consider commands 'n' and 'q', but any other character
4340159                 // can be introduced, to let show the next page.
4340160                 //
4340161                 switch (c)
4340162                   {
4340163                     case 'N':
4340164                     case 'n':
```

```
4340165                    //
4340166                    // Go to the next file, if any.
4340167                    //
4340168                    fclose (fp);
4340169                    fp = NULL;
4340170                    loop = 0;
4340171                    break;
4340172                case 'Q':
4340173                case 'q':
4340174                    //
4340175                    // Quit. But must erase the `--More--' prompt.
4340176                    //
4340177                    printf ("\b \b\b \b\b \b\b \b\b \b");
4340178                    printf ("\b \b\b \b\b \b\b \b");
4340179                    fclose (fp);
4340180                    return (0);
4340181              }
4340182            //
4340183            // Backspace to overwrite `--More--' and the character
4340184            // pressed.
4340185            //
4340186            printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
4340187            //
4340188            // Reset line/column counters.
4340189            //
4340190            column = 1;
4340191            line = 1;
4340192          }
4340193        //
4340194        // Close the file pointer if it is still open.
4340195        //
4340196        if (fp != NULL)
4340197          {
4340198            fclose (fp);
4340199          }
4340200      }
4340201    //
4340202    return (0);
4340203 }
4340204 //----------------------------------------------------------------------
4340205 static void
4340206 usage (void)
4340207 {
4340208     fprintf (stderr, "Usage: more FILE...\n");
4340209 }
```

102.1.21 applic/mount.c

Si veda la sezione 98.4.

```
4350001 #include <unistd.h>
4350002 #include <stdlib.h>
4350003 #include <sys/stat.h>
4350004 #include <sys/types.h>
4350005 #include <fcntl.h>
4350006 #include <errno.h>
4350007 #include <signal.h>
4350008 #include <stdio.h>
4350009 #include <sys/wait.h>
4350010 #include <stdio.h>
4350011 #include <string.h>
4350012 #include <limits.h>
4350013 #include <sys/os16.h>
4350014 //----------------------------------------------------------------------
4350015 static void usage (void);
4350016 //----------------------------------------------------------------------
4350017 int
4350018 main (int argc, char *argv[], char *envp[])
4350019 {
4350020     int     options;
4350021     int     status;
4350022     //
4350023     //
4350024     //
4350025     if (argc < 3 || argc > 4)
4350026       {
4350027         usage ();
4350028         return (1);
4350029       }
4350030     //
4350031     // Set options.
4350032     //
4350033     if (argc == 4)
4350034       {
4350035         if      (strcmp (argv[3], "rw") == 0)
4350036           {
4350037             options = MOUNT_DEFAULT;
4350038           }
4350039         else if (strcmp (argv[3], "ro") == 0)
4350040           {
4350041             options = MOUNT_RO;
4350042           }
4350043         else
4350044           {
4350045             printf ("Invalid mount option: only \"ro\" or \"rw\" "
```

```
4350046                          "are allowed\n");
4350047                  return (2);
4350048                }
4350049          }
4350050        else
4350051          {
4350052            options = MOUNT_DEFAULT;
4350053          }
4350054        //
4350055        // System call.
4350056        //
4350057        status = mount (argv[1], argv[2], options);
4350058        if (status != 0)
4350059          {
4350060            perror (NULL);
4350061            return (2);
4350062          }
4350063        //
4350064        return (0);
4350065    }
4350066    //----------------------------------------------------------------------
4350067    static void
4350068    usage (void)
4350069    {
4350070        fprintf (stderr, "Usage: mount DEVICE MOUNT_POINT "
4350071                          "[MOUNT_OPTIONS]\n");
4350072    }
```

102.1.22 applic/ps.c

«

Si veda la sezione 92.17.

```
4360001    #include <kernel/proc.h>
4360002    #include <unistd.h>
4360003    #include <stdio.h>
4360004    #include <fcntl.h>
4360005    #include <unistd.h>
4360006    #include <stdlib.h>
4360007    //----------------------------------------------------------------------
4360008    void
4360009    print_proc_head (void)
4360010    {
4360011        printf (
4360012    "pp  p pg                                                            \n"
4360013    "id id rp  tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name\n"
4360014               );
4360015    }
4360016    //----------------------------------------------------------------------
```

```
4360017 void
4360018 print_proc_pid (proc_t *ps, pid_t pid)
4360019 {
4360020     char  stat;
4360021     switch (ps->status)
4360022       {
4360023         case PROC_EMPTY   : stat = '-'; break;
4360024         case PROC_CREATED : stat = 'c'; break;
4360025         case PROC_READY   : stat = 'r'; break;
4360026         case PROC_RUNNING : stat = 'R'; break;
4360027         case PROC_SLEEPING: stat = 's'; break;
4360028         case PROC_ZOMBIE  : stat = 'z'; break;
4360029         default           : stat = '?'; break;
4360030       }
4360031 
4360032     printf ("%2i %2i %2i %04x %4i %4i %4i %02i.%02i %c %05lx %04x ",
4360033               (unsigned int) ps->ppid,
4360034               (unsigned int) pid,
4360035               (unsigned int) ps->pgrp,
4360036               (unsigned int) ps->device_tty,
4360037               (unsigned int) ps->uid,
4360038               (unsigned int) ps->euid,
4360039               (unsigned int) ps->suid,
4360040               (unsigned int) ((ps->usage / CLOCKS_PER_SEC) / 60),
4360041               (unsigned int) ((ps->usage / CLOCKS_PER_SEC) % 60),
4360042               stat,
4360043               (unsigned long int) ps->address_i,
4360044               (unsigned int) ps->size_i);
4360045 
4360046     printf ("%05lx %04x %04x %s",
4360047               (unsigned long int) ps->address_d,
4360048               (unsigned int) ps->size_d,
4360049               (unsigned int) ps->sp,
4360050                                   ps->name);
4360051 
4360052     printf ("\n");
4360053 }
4360054 //----------------------------------------------------------------------
4360055 int
4360056 main (void)
4360057 {
4360058     pid_t   pid;
4360059     proc_t *ps;
4360060     int     fd;
4360061     ssize_t size_read;
4360062     char    buffer[sizeof (proc_t)];
4360063 
4360064     fd = open ("/dev/kmem_ps", O_RDONLY);
```

```
4360065     if (fd < 0)
4360066       {
4360067         perror ("ps: cannot open \"/dev/kmem_ps\"");
4360068         exit (0);
4360069       }
4360070
4360071     print_proc_head ();
4360072     for (pid = 0; pid < PROCESS_MAX; pid++)
4360073       {
4360074         lseek (fd, (off_t) pid, SEEK_SET);
4360075         size_read = read (fd, buffer, sizeof (proc_t));
4360076         if (size_read < sizeof (proc_t))
4360077           {
4360078             printf ("ps: cannot read \"/dev/kmem_ps\" pid %i", pid);
4360079             perror (NULL);
4360080             continue;
4360081           }
4360082         ps = (proc_t *) buffer;
4360083         if (ps->status > 0)
4360084           {
4360085             ps->name[PATH_MAX-1] = 0; // Terminated string.
4360086             print_proc_pid (ps, pid);
4360087           }
4360088       }
4360089
4360090     close (fd);
4360091     return (0);
4360092 }
```

102.1.23 applic/rm.c

«

Si veda la sezione 92.18.

```
4370001 #include <fcntl.h>
4370002 #include <sys/stat.h>
4370003 #include <stddef.h>
4370004 #include <unistd.h>
4370005 #include <errno.h>
4370006 //----------------------------------------------------------------------
4370007 static void usage (void);
4370008 //----------------------------------------------------------------------
4370009 int
4370010 main (int argc, char *argv[], char *envp[])
4370011 {
4370012     int         a;                // Argument index.
4370013     int         status;
4370014     struct stat file_status;
4370015     //
```

```
4370016     // No options are known, but at least an argument must be given.
4370017     //
4370018     if (argc < 2)
4370019       {
4370020         usage ();
4370021         return (1);
4370022       }
4370023     //
4370024     // Scan arguments.
4370025     //
4370026     for(a = 1; a < argc; a++)
4370027       {
4370028         //
4370029         // Verify if the file exists.
4370030         //
4370031         if (stat(argv[a], &file_status) != 0)
4370032           {
4370033             fprintf (stderr, "File \"%s\" does not exist!\n",
4370034                      argv[a]);
4370035             continue;
4370036           }
4370037         //
4370038         // File exists: check the file type.
4370039         //
4370040         if (S_ISDIR (file_status.st_mode))
4370041           {
4370042             fprintf (stderr, "Cannot remove directory \"%s\"!\n",
4370043                      argv[a]);
4370044             continue;
4370045           }
4370046         //
4370047         // Can remove it.
4370048         //
4370049         status = unlink (argv[a]);
4370050         if (status != 0)
4370051           {
4370052             perror (NULL);
4370053             return (2);
4370054           }
4370055       }
4370056     return (0);
4370057 }
4370058 //----------------------------------------------------------------------
4370059 static void
4370060 usage (void)
4370061 {
4370062     fprintf (stderr, "Usage: rm FILE...\n");
```

```
4370063 }
```

102.1.24 applic/shell.c

«

Si veda la sezione 92.19.

```
4380001 #include <unistd.h>
4380002 #include <stdlib.h>
4380003 #include <sys/stat.h>
4380004 #include <sys/types.h>
4380005 #include <fcntl.h>
4380006 #include <errno.h>
4380007 #include <unistd.h>
4380008 #include <signal.h>
4380009 #include <stdio.h>
4380010 #include <sys/wait.h>
4380011 #include <stdio.h>
4380012 #include <string.h>
4380013 #include <limits.h>
4380014 #include <sys/os16.h>
4380015 //----------------------------------------------------------------------
4380016 #define PROMPT_SIZE     16
4380017 //----------------------------------------------------------------------
4380018 static void sh_cd       (int argc, char *argv[]);
4380019 static void sh_pwd      (int argc, char *argv[]);
4380020 static void sh_umask    (int argc, char *argv[]);
4380021 //----------------------------------------------------------------------
4380022 int
4380023 main (int argc, char *argv[], char *envp[])
4380024 {
4380025   char    buffer_cmd[ARG_MAX/2];
4380026   char   *argv_cmd[ARG_MAX/16];
4380027   char    prompt[PROMPT_SIZE];
4380028   uid_t   uid;
4380029   int     argc_cmd;
4380030   pid_t   pid_cmd;
4380031   pid_t   pid_dead;
4380032   int     status;
4380033   //
4380034   //
4380035   //
4380036   uid = geteuid ();
4380037   //
4380038   // Load processes, reading the keyboard.
4380039   //
4380040   while (1)
4380041     {
4380042       if (uid == 0)
```

```
4380043             {
4380044               strncpy (prompt, "# ", PROMPT_SIZE);
4380045             }
4380046           else
4380047             {
4380048               strncpy (prompt, "$ ", PROMPT_SIZE);
4380049             }
4380050           //
4380051           input_line (buffer_cmd, prompt, (ARG_MAX/2), INPUT_LINE_ECHO);
4380052           //
4380053           // Clear `argv_cmd[]';
4380054           //
4380055           for (argc_cmd = 0; argc_cmd < (ARG_MAX/16); argc_cmd++)
4380056             {
4380057               argv_cmd[argc_cmd] = NULL;
4380058             }
4380059           //
4380060           // Initialize the command scan.
4380061           //
4380062           argv_cmd[0] = strtok (buffer_cmd, " \t");
4380063           //
4380064           // Verify: if the input is not valid, loop again.
4380065           //
4380066           if (argv_cmd[0] == NULL)
4380067             {
4380068               continue;
4380069             }
4380070           //
4380071           // Find the arguments.
4380072           //
4380073           for (argc_cmd = 1;
4380074                argc_cmd < ((ARG_MAX/16)-1) && argv_cmd[argc_cmd-1] != NULL;
4380075                argc_cmd++)
4380076             {
4380077               argv_cmd[argc_cmd] = strtok (NULL, " \t");
4380078             }
4380079           //
4380080           // If there are too many arguments, show a message and continue.
4380081           //
4380082           if (argv_cmd[argc_cmd-1] != NULL)
4380083             {
4380084               errset (E2BIG);              // Argument list too long.
4380085               perror (NULL);
4380086               continue;
4380087             }
4380088           //
4380089           // Correct the value for `argc_cmd', because actually
4380090           // it counts also the NULL element.
```

```
4380091         //
4380092         argc_cmd--;
4380093         //
4380094         // Verify if it is an internal command.
4380095         //
4380096         if      (strcmp (argv_cmd[0], "exit") == 0)
4380097           {
4380098             return (0);
4380099           }
4380100         else if (strcmp (argv_cmd[0], "cd") == 0)
4380101           {
4380102             sh_cd (argc_cmd, argv_cmd);
4380103             continue;
4380104           }
4380105         else if (strcmp (argv_cmd[0], "pwd") == 0)
4380106           {
4380107             sh_pwd (argc_cmd, argv_cmd);
4380108             continue;
4380109           }
4380110         else if (strcmp (argv_cmd[0], "umask") == 0)
4380111           {
4380112             sh_umask (argc_cmd, argv_cmd);
4380113             continue;
4380114           }
4380115         //
4380116         // It should be a program to run.
4380117         //
4380118         pid_cmd = fork ();
4380119         if (pid_cmd == -1)
4380120           {
4380121             printf ("%s: cannot run command", argv[0]);
4380122             perror (NULL);
4380123           }
4380124         else if (pid_cmd == 0)
4380125           {
4380126             execvp (argv_cmd[0], argv_cmd);
4380127             perror (NULL);
4380128             exit (0);
4380129           }
4380130         while (1)
4380131           {
4380132             pid_dead = wait (&status);
4380133             if (pid_dead == pid_cmd)
4380134               {
4380135                 break;
4380136               }
4380137           }
4380138         printf ("pid %i terminated with status %i.\n",
```

```
4380139                 (int) pid_dead, status);
4380140     }
4380141 }
4380142 //----------------------------------------------------------------------
4380143 static void
4380144 sh_cd (int argc, char *argv[])
4380145 {
4380146   int status;
4380147   //
4380148   if (argc != 2)
4380149     {
4380150       errset (EINVAL);                  // Invalid argument.
4380151       perror (NULL);
4380152       return;
4380153     }
4380154   //
4380155   status = chdir (argv[1]);
4380156   if (status != 0)
4380157     {
4380158       perror (NULL);
4380159     }
4380160   return;
4380161 }
4380162 //----------------------------------------------------------------------
4380163 static void
4380164 sh_pwd (int argc, char *argv[])
4380165 {
4380166   char  path[PATH_MAX];
4380167   void *pstatus;
4380168   //
4380169   if (argc != 1)
4380170     {
4380171       errset (EINVAL);                  // Invalid argument.
4380172       perror (NULL);
4380173       return;
4380174     }
4380175   //
4380176   // Get the current directory.
4380177   //
4380178   pstatus = getcwd (path, (size_t) PATH_MAX);
4380179   if (pstatus == NULL)
4380180     {
4380181       perror (NULL);
4380182     }
4380183   else
4380184     {
4380185       printf ("%s\n", path);
4380186     }
```

```
4380187     return;
4380188   }
4380189   //----------------------------------------------------------------------
4380190   static void
4380191   sh_umask (int argc, char *argv[])
4380192   {
4380193     sysmsg_uarea_t msg;
4380194     char           *m;          // Index inside the umask octal string.
4380195     int             mask;
4380196     int             digit;
4380197     //
4380198     if (argc > 2)
4380199       {
4380200         errset (EINVAL);                  // Invalid argument.
4380201         perror (NULL);
4380202         return;
4380203       }
4380204     //
4380205     // If no argument is available, the umask is shown, with a direct
4380206     // system call.
4380207     //
4380208     if (argc == 1)
4380209       {
4380210         sys (SYS_UAREA, &msg, (sizeof msg));
4380211         printf ("%04o\n", msg.umask);
4380212         return;
4380213       }
4380214     //
4380215     // Get the mask: must be the first argument.
4380216     //
4380217     for (mask = 0, m = argv[1]; *m != 0; m++)
4380218       {
4380219         digit = (*m - '0');
4380220         if (digit < 0 || digit > 7)
4380221           {
4380222             errset (EINVAL);              // Invalid argument.
4380223             perror (NULL);
4380224             return;
4380225           }
4380226         mask = mask * 8 + digit;
4380227       }
4380228     //
4380229     // Set the umask and return.
4380230     //
4380231     umask (mask);
4380232     return;
4380233   }
```

102.1.25 applic/touch.c

Si veda la sezione 92.20.

```
4390001 #include <fcntl.h>
4390002 #include <sys/stat.h>
4390003 #include <utime.h>
4390004 #include <stddef.h>
4390005 #include <unistd.h>
4390006 #include <errno.h>
4390007 //----------------------------------------------------------------------
4390008 static void usage (void);
4390009 //----------------------------------------------------------------------
4390010 int
4390011 main (int argc, char *argv[], char *envp[])
4390012 {
4390013     int         a;              // Argument index.
4390014     int         status;
4390015     struct stat file_status;
4390016     //
4390017     // No options are known, but at least an argument must be given.
4390018     //
4390019     if (argc < 2)
4390020       {
4390021         usage ();
4390022         return (1);
4390023       }
4390024     //
4390025     // Scan arguments.
4390026     //
4390027     for(a = 1; a < argc; a++)
4390028       {
4390029         //
4390030         // Verify if the file exists, through the return value of
4390031         // 'stat()'. No other checks are made.
4390032         //
4390033         if (stat(argv[a], &file_status) == 0)
4390034          {
4390035             //
4390036             // File exists: should be updated the times.
4390037             //
4390038             status = utime (argv[a], NULL);
4390039             if (status != 0)
4390040               {
4390041                 perror (NULL);
4390042                 return (2);
4390043               }
4390044           }
4390045         else
```

```
4390046             {
4390047               //
4390048               // File does not exist: should be created.
4390049               //
4390050               status = open (argv[a], O_WRONLY|O_CREAT|O_TRUNC, 0666);
4390051               //
4390052               if (status >= 0)
4390053                 {
4390054                   //
4390055                   // Here, the variable 'status' is the file
4390056                   // descriptor to be closed.
4390057                   //
4390058                   status = close (status);
4390059                   if (status != 0)
4390060                     {
4390061                       perror (NULL);
4390062                       return (3);
4390063                     }
4390064                 }
4390065               else
4390066                 {
4390067                   perror (NULL);
4390068                   return (4);
4390069                 }
4390070             }
4390071         }
4390072     return (0);
4390073 }
4390074 //----------------------------------------------------------------------
4390075 static void
4390076 usage (void)
4390077 {
4390078     fprintf (stderr, "Usage: touch FILE...\n");
4390079 }
```

102.1.26 applic/tty.c

Si veda la sezione 92.21.

```
4400001 #include <fcntl.h>
4400002 #include <sys/stat.h>
4400003 #include <utime.h>
4400004 #include <stddef.h>
4400005 #include <unistd.h>
4400006 #include <errno.h>
4400007 #include <sys/os16.h>
4400008 #include <sys/types.h>
4400009 //----------------------------------------------------------------------
```

```
4400010 static void     usage (void);
4400011 //----------------------------------------------------------------------
4400012 int
4400013 main (int argc, char *argv[], char *envp[])
4400014 {
4400015   int             dev_minor;
4400016   struct stat     file_status;
4400017   //
4400018   // No options and no arguments.
4400019   //
4400020   if (argc > 1)
4400021     {
4400022       usage ();
4400023       return (1);
4400024     }
4400025   //
4400026   // Verify the standard input.
4400027   //
4400028   if (fstat (STDIN_FILENO, &file_status) == 0)
4400029     {
4400030       if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
4400031         {
4400032           dev_minor = minor (file_status.st_rdev);
4400033           //
4400034           // If minor is equal to 0xFF, it is '/dev/console'
4400035           // that is not a controlling terminal, but just
4400036           // a reference for the current virtual console.
4400037           //
4400038           if (dev_minor < 0xFF)
4400039             {
4400040               printf ("/dev/console%i\n", dev_minor);
4400041             }
4400042         }
4400043     }
4400044   else
4400045     {
4400046       perror ("Cannot get standard input file status");
4400047       return (2);
4400048     }
4400049   //
4400050   return (0);
4400051 }
4400052
4400053 //----------------------------------------------------------------------
4400054 static void
4400055 usage (void)
4400056 {
4400057   fprintf (stderr, "Usage: tty\n");
```

```
4400058 }
```

102.1.27 applic/umount.c

«

Si veda la sezione 98.4.

```
4410001 #include <unistd.h>
4410002 #include <stdlib.h>
4410003 #include <sys/stat.h>
4410004 #include <sys/types.h>
4410005 #include <fcntl.h>
4410006 #include <errno.h>
4410007 #include <signal.h>
4410008 #include <stdio.h>
4410009 #include <sys/wait.h>
4410010 #include <stdio.h>
4410011 #include <string.h>
4410012 #include <limits.h>
4410013 #include <sys/os16.h>
4410014 //----------------------------------------------------------------------
4410015 static void usage (void);
4410016 //----------------------------------------------------------------------
4410017 int
4410018 main (int argc, char *argv[], char *envp[])
4410019 {
4410020     int     status;
4410021     //
4410022     // One argument is mandatory.
4410023     //
4410024     if (argc != 2)
4410025       {
4410026         usage ();
4410027         return (1);
4410028       }
4410029     //
4410030     // System call.
4410031     //
4410032     status = umount (argv[1]);
4410033     if (status != 0)
4410034       {
4410035         perror (argv[1]);
4410036         return (2);
4410037       }
4410038     //
4410039     return (0);
4410040 }
4410041 //----------------------------------------------------------------------
4410042 static void
```

```
4410043 usage (void)
4410044 {
4410045     fprintf (stderr, "Usage: umount MOUNT_POINT\n");
4410046 }
```

Indice analitico del volume